Depot: A Tool for Managing Software Environments

Wallace Colyer & Walter Wong – Carnegie Mellon University

ABSTRACT

`Depot` is a software management tool which provides a simple, yet flexible, mechanism for maintaining third party and locally developed software in large heterogeneous computing environments. `Depot` integrates separately maintained software packages, known as collections, into a common directory hierarchy consisting of a union of all the collections. This common directory is defined as the software environment. A set of configuration options manages interactions and intersections between collections in the environment.

`Depot` facilitates the introduction, update, and removal of collections in a software environment. Custom environments and complete test environments can be easily created for individual machines or for sets of machines. Collections with unexpected problems can either be replaced with previous versions or removed. Individual collections or files can be moved from remote filesystems to the local disks of workstations without the concern that those files may become outdated. All this is achieved with minimal wasted disk space and administrative overhead.

Introduction

The installation and maintenance of application software on UNIX platforms has traditionally been a difficult and time consuming process. Many difficulties result from inadequate software release and environment control tools. The situation is aggravated by a complete lack of industry standards, the common use of hard coded paths for file dependencies, and unreasonable assumptions that many software providers make of the installation environment.

The emergence and popularity of distributed computing has compounded the management problem. A large heterogeneous environment, with thousands of workstations and hundreds of software packages, aggravates the existing problems and adds new ones which must be overcome.

To properly manage a software environment several issues must be resolved. An inventory must be maintained containing the origin of all the components of the environment. Software must be thoroughly tested independently, as well as in the destination environments. If a critical problem escapes the testing process, the software environment must be smoothly restorable to a previous working state.

In a distributed software environment, there is a need to distribute and install software on remote machines with different architectures, customizations and configurations. The procedures required must minimize the workload of the system administrators.

Many solutions that manage a distributed software environment often bring back load and availability problems of timesharing systems by increasing the dependence on centrally maintained services. To prevent this, workstations should be able to locally cache commonly used files, as well as maintain a core set of functionality in case of server or network failures.

A software release management system in complex environments should handle the following issues:

- Distribution
- Installation
- Customization
- Testing
- Removal and restoration

By segregating the environment into discrete manageable objects, it is possible to address all of these issues. These objects can then be layered to create the user visible environment. Thus, the environment can be looked at as either a whole or in parts.

Motivation

In the past, the software maintainers of the Andrew system installed software, following the general UNIX philosophy, directly into the **/usr/local** tree. As the number of applications multiplied, the maintenance process became increasingly difficult. For example, because no records were kept of the files that were installed with a software package, often outdated files were left in the system, wasting valuable disk space.

Another problem occurred when two applications had files with the same name installed in the same directory; the conflicting file would be overwritten during the installation process. This would lead to all sorts of subtle problems, especially if the files were significantly different between the two applications.

When our software manager began posting lists of hundreds of files and asking if anyone knew whether they still belonged in **/usr/local**, it became apparent that we had a serious problem. We identified four key components necessary to solving this problem: independence, integration, mobility and simplicity.

Independence

The problem of keeping track of software can be solved by separating sets of related software into independent directory hierarchies called *collections*. The collection abstraction reduces the complexity of the software environment by creating smaller, well defined, working groups. Each collection is kept in separate locations, so it is simple to determine the origin of its files. This facilitates finding and reporting problems, as well as cleaning up the environment when updates occur or when the software package becomes obsolete. Furthermore, a complete software package can be distributed or shared by simply specifying the path to the collection.

`The Depot` [Manh90], developed at the National Institute of Standards and Technology (NIST), splits applications into different collections; however, no integration is done. To access files in a collection, the user must be aware of this separation and have a very long list of items on his path. We considered this approach to be too cumbersome in the Andrew environment.

Many problems had been encountered when making changes to the search paths of over 10,000 users of our system, many of whom access our filesystem from departmental computing facilities where we do not have administrative access. These departments wish to import a single directory hierarchy to their machines, such as **/usr/local**, in order to use our software. It is convenient for there to be a single path to all software in that hierarchy, i.e. **/usr/local/bin**. While it is possible to have configuration files which all users access to set the paths, it is difficult to keep those files up to date and to ensure users in other departments will use these centrally maintained site configuration files.

Integration

Not only is it necessary to maintain independent collections for the sake of administrative convenience, it is also important to unify the collections into a single directory hierarchy. This helps the users understand the environment as they do not have to look in many different places for system software. This provides, as well, a way for applications to share common directories.

Many applications need to share directories where common files are kept. Index files are often kept that must be updated whenever any files change in these directories. Two common examples of this are: `X11` fonts and `man` pages. With total separation, it becomes increasingly difficult to seamlessly integrate the environment.

Maintaining independent collections does not, however, address the problem of two applications installing binaries with the same name -- path conflicts and ordering problems still remain.

These problems can be addressed by integrating the independent collections into a common directory hierarchy and forcing conflict resolution.

Mobility

The most obvious reason for wanting to efficiently move software in and out of environments is for testing. If environments can be created without regard to the actual location of the collections, software can be tested by creating a duplicate destination environment. It should be possible to generate a software environment that is identical to the current released environment, with the exception of the collection to be tested. If major problems were uncovered, it should be simple to restore the old environment quickly.

Distributed filesystems illustrate the concept of mobility. Collections should be able to move between the remote filesystem and the local disk of the client workstation. Collections on the local disk should be updated when new versions come out as transparently as possible. Rarely used applications may be stored on the remote filesystem, thus conserving local disk space. Commonly used or important applications may be stored on the local disk, thereby increasing access speed and availability. Regardless, the actual location of the software should be transparent to the end user.

While other systems, such as `Xhier` [Sell91], have recognized the need for independence and integration, no package that we examined addressed the issue of mobility. Most provide only one environment, for example, `Xhier`'s **/software** and NIST's `Depot`'s **/depot**. Ideally, multiple environments would be possible. We have found it desirable to have an environment for fully supported software, **/usr/local**, and another for "unsupported but useful" software, **/usr/contributed**. Moreover, it should be possible to easily move collections from **/usr/contributed** to **/usr/local** and vice versa.

Simplicity

Paul Anderson [Ande91] described a method of tracking software by tagging files in each collection with a unique UNIX userid (uid). This approach satisfied the independence, integration, and, to an extent, the mobility requirements. However, the process has a good deal of complexity, since developers are required to do extra work to utilize the system. Additional tools are required to track and maintain the collections. Furthermore, in a decentralized distributed environment, password files must remain homogeneous across all machines.

Alternatively, simplicity and understandability could be maintained by having each collection imported into the software environment by using its own directory hierarchy. For example, a file placed in the **bin** directory of a collection should appear in the **bin** directory of the environment. The software installer should only need to determine the desired directory hierarchy. When the collection appears in the environment it will reflect that hierarchy. The environment maintainer should only need to decide which collections to integrate into the environment and how to resolve any conflicts, or when two collections try to install the same file in the same location.

Additionally, the system can be kept simple by not incorporating the distribution mechanism into the program. For example, a distributed filesystem, such as `AFS`[1] [Saty85], or standard software distribution tools such as `rdist` and `SUP` [Shaf88] may be used. Distribution may become important for certain classes of machines, such as laptops, and other computers connected by slow or unreliable network connections, but any distribution solution should still be external.

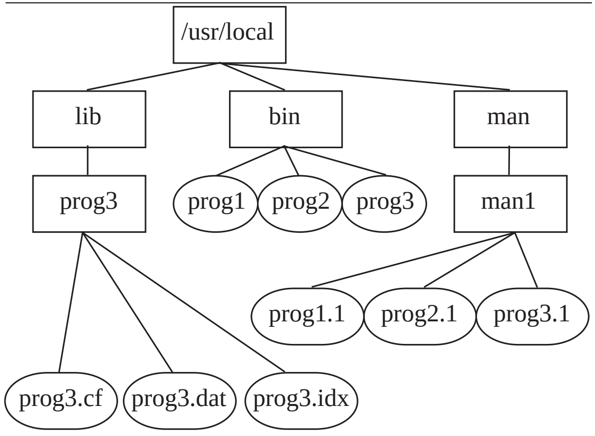

Figure 1: Simple **/usr/local** Environment

Implementation

`Depot` creates a system which requires relatively little work to setup and maintain multiple environments, to easily allow software to be quickly installed and backed out, and to allow individual workstations to be customized. This is done in a manner that minimizes the overall complexity of maintaining large software environments. `Depot` achieves the goals of independence, integration, mobility and simplicity. The system is implemented by integrating multiple independent collections into a single directory hierarchy and allowing specific customizations via configuration files.

[1]The Andrew Filesystem (`AFS`) is a scalable distributed filesystem available from the Transarc Corporation

With each invocation, `depot` processes a single software environment. The software environment starts with a specified directory hierarchy and encompasses everything within it, including subdirectories. Figure 1 shows a simple **/usr/local** environment.

`Depot` defines the environment as the union of a set of software collections. Figure 2 shows a way collections can be stored in the `depot` framework. In the environment, the **depot** directory is special. This directory stores the database and configuration files and, in this case, stores the collections.

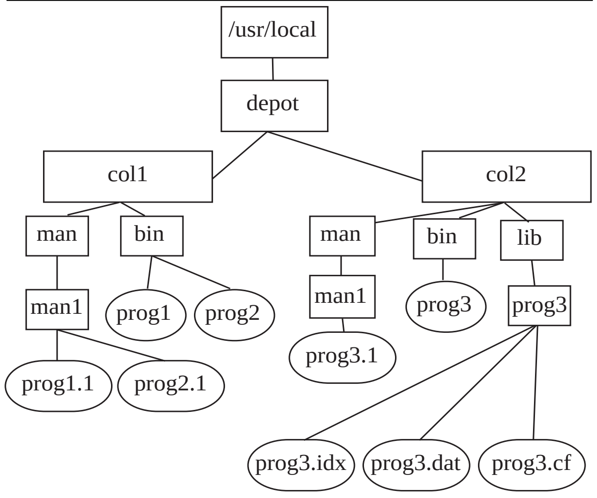

Figure 2: Collections

The environment is customized through a set of configuration options. These options determine which collections will be integrated into the environment and how they will be integrated.

Collections can be integrated into an environment in several ways:

- By listing specific collections and the paths to their location
- By providing search paths where the first instance of each collection within the path will be used
- By placing the collections in the **depot** directory of the environment, as shown in figure 2
- Or by using a combination of these methods

Before integrating the collections into an environment, `depot` verifies that the environment is consistent. Files in the in the environment that do not belong to any collection or are not marked as special files are deleted. `Depot` will then check to see if any of the collections on its paths have been changed, added or deleted. All the new collections will be added to the environment, all removed collections will be deleted, and any necessary changes for modified collections will be made.

As `depot`'s last action, other binaries can be run if a specified collection changes. This addresses the issue where special files need to be generated.

For example, often multiple collections install files into a common directory. Index files are often kept of the contents of these directory, such as **fonts.dir** in the X11 font directories. Future versions of depot will have configuration options to run the commands whenever the directory structure changes.

In the integration process, there is the chance that files, from different collections, will export to the same place. For example, if two collections both have the file **bin/foo**, then a *conflict* has occurred between the two collections. Depot will exit at this time. To resolve the conflict, the environment maintainer can specify that one collection is to override the other. Alternatively, the environment maintainer may request the collection maintainer to move files or directory hierarchies by changing the collections or by using collection specific configuration files.

There are two ways a collection can be integrated: copying or linking. For collections that are linked, symbolic links are made from the environment to their location in the collections specific directory. To reduce the overhead of the links, they are made at the directory level wherever possible. With the copy option, every file and directory is copied into the target environment.

In earlier versions, depot was only able to operate at the collection level. There was no way to copy or link individual files or directories; the entire collection was either copied or linked. This strict separation, with no single file operations, proved to be too restricting. Target specific options now permit individual files or directories to be copied, linked, deleted or ignored, regardless of the collection of origin. For example, several collections may install fonts into the **lib/X11/fonts** directory, and the environment maintainer may wish them to always be copied, regardless of the collection they came from. On the other hand, the environment maintainer may choose to link all the files integrated into the **man** or **doc** directories to conserve space. Since these options work only by changing the behavior when a file is mapped out of a collection into the target directory, and they do not modify the resulting structure, the sanctity of the collection is maintained, and a great deal of flexibility is achieved for the environment maintainer.

Figure 3 presents an environment integrated by using symbolic links. This environment is generated by depot from the collections in Figure 2 and would present the same structure, to the user, as the one shown in Figure 1. In this case, the environment was generated by symbolic links which are represented by the shaded objects. The arrows point to the actual location of the files or directories.

When creating an environment with symbolic links depot performs link optimization in order to link at the highest possible level of the collection hierarchy. This reduces the number of symbolic links depot must make. Figure 3 shows this process. Since the **lib** directory only exists in *col2*, depot links **/usr/local/lib** directly to the **lib** directory in the *col2* collection. If another collection later introduces a file into a directory which has been optimized at a higher level, the link will be

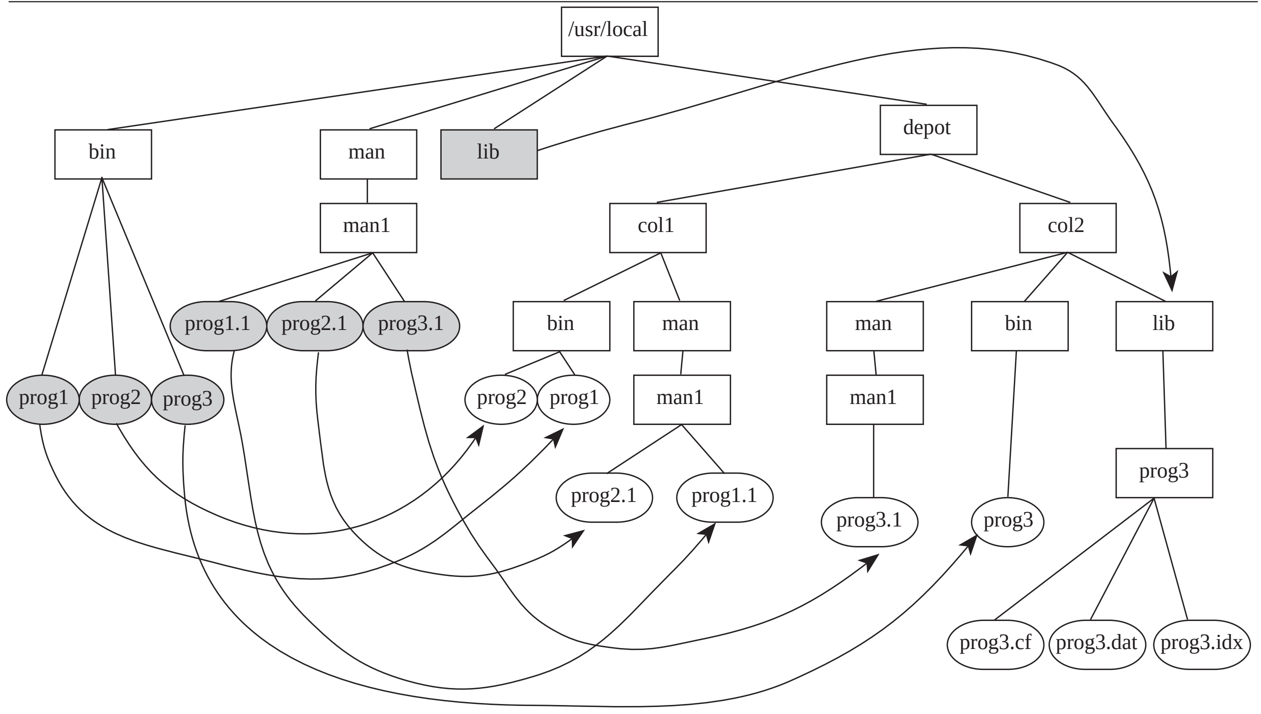

Figure 3: Symbolic Link Environment

removed and all the files at the lower level will be linked in. It will again attempt to link any subdirectories of the previously optimized directory new links are made. Many collections (e.g., the Framemaker publishing package), have hierarchies of thousands of files into which it is unlikely any other application would ever introduce files.

In our environment, the software environment is first integrated on a shared filesystem. Most workstations may just access the environment in that manner. However, workstations with more local storage may move the environment to their local disk. This creates the situation where the client should be updated whenever software is updated in the shared filesystem. For example, a symbolic link could exist from the local disk to a file in a collection on the shared filesystem. If, the collection maintainer changes the collection by removing the file, there will be the case where a symbolic link is pointing at a nonexistent file.

To resolve the consistency problems, the client workstation must either run depot immediately, or there must be a way for the local workstation environment to remain consistent and fully functional until a scheduled run of depot occurs. Requiring all participating machines to run depot simultaneously in a large workstation installation neither feasible nor practical. To allow workstations to run depot on their own time frame, we added the concept of depot version numbers. When a new version of a software package is released to the environment, it is mounted with a higher version number. The highest version is selected and integrated into the environment. A reasonable number of versions are kept so no collections will be erased before a workstation has an opportunity to run depot. Thus, functionality and consistency of the environment is preserved.

By integrating multiple independent collections into a single environment, depot achieves independence and integration. The search paths, version numbers, and different updating strategies provide mobility by allowing the integration of new or different versions of a software package from different locations. Finally, the mirroring of directory hierarchies and simple configuration options are easy for administrators and software developers to understand and use, thereby achieving the goal of simplicity.

Limitations

Depot only operates inside a single environment at one time. Software managed in **/usr/local** cannot be moved by depot outside of **/usr/local**. Software or files that need to be copied into the operating system areas will require another program to do so. There also lacks a mechanism to scan the entire environment for conflicts. This makes building the environment for the very first time a somewhat longer and more tedious task.

Currently, depot is somewhat inefficient at dealing with very large environments. The time to search its databases and to *stat*(2) source directories for changed collections increases undesirable as the environment grows. Some performance enhancements have been made by introducing code specific to AFS volumes[2] [Side86], but these have not been sufficient. A network server or hint files, containing modification dates of collections and information about their tree structure, may be needed. A complete rewrite of the database and customization handling routines is planned.

Some additional tools are required for distribution and for detailed tracking of software in the environments. Colyer, et. al., [Coly92] provides an overview of the tools used with Depot to manage the Andrew Software Environment. Mark Held [Held92] provides a more detailed explanation of the environment. Also, as a result of our AFS environment, the issue of architecture differences is not addressed by depot. This issue must be handled by the distribution system.

We are planning for depot to replace package [Youn85], our current host configuration tool, and make depot a workstation manager. The environment would be the operating system of the workstation. Each operating system release would be a collection where minor release levels would override the major release. Layered operating system products would also be collections in the environment. In addition to this, a hierarchy of overrides are also required where "Andrew" changes would override operating system defaults. Further, departmental changes would override "Andrew" changes and finally local workstation changes, with the highest priority, would override all other changes.

Conclusion

In the Andrew environment, where depot was developed, it would be unthinkable to return to the situation such a tool was available. Installations were lengthy, error prone processes. Often the installation of a new application would break previous applications. There was no smooth way to restore the environment to a previous state. Even though numerous man hours were put into maintaining the environment, the system was essentially in a state of anarchy.

Today, even with multiple environments, software can be easily installed and removed from the system. Individual workstations can be customized to achieve a degree of network independence with minimal effort by the central staff or workstations owners. Much of depot's success can be attributed to the four factors discussed earlier:

[2]Volumes are containers of UNIX filesystems, similar to disk partitions. They are the administrative unit of AFS.

independence, integration, mobility and simplicity. The concept of combining independence and integration provided the manageability we needed without sacrificing the consistency that users demand. Mobility gives us flexibility in configuration and testing. Finally, the simplicity has made it popular with developers and allows us to integrate `depot` with other tools, rather than trying to make `depot` a "kitchen sink" tool. `Depot` has proved to be a flexible mechanism for maintaining our software environment.

Acknowledgments

Without the considerable work done by a large number of people `depot` would not have been possible. Many more people were influential in the creation of `depot` and the software management procedures along side it.

In 1988, `depot` was born during a set of software management brain storming sessions attended by Wallace Colyer, Mark Held, Ted McCabe, and David VanRyzin. Each member of this group contributed to the creation of `depot`. There were many useful insights gained from previous software management strategies developed in conjunction with the Information Technology Center (ITC) and groups within the Academic Services division at Carnegie Mellon University. Mike Accetta and other members of the School of Computer Science were very helpful during our initial consultations in explaining the strengths and weaknesses of their **/usr/misc** software management system. **/usr/misc** provided the initial ideas for the creation of `depot`. The original prototype was written in `perl` [Wall91] by Wallace Colyer in 1989. It has since been rewritten in C by was written by Sohan C. Ramakrishna-Pillai.

We would like to thank Terilyn Gillespie and Dawn Neuhart for helping to, once and for all, finish this paper.

A final set of thanks goes to Mark Held, who is leading the effort of maintaining our local and third party software. He also undertook the enormous project to migrate all our software into the new architecture and has produced an excellent software development environment based on the framework provided by `depot`.

Availability

`Depot` is available via anonymous ftp from *export.acs.cmu.edu* [128.2.35.66] in **/pub/depot**. `Depot` is also available via AFS in **/afs/andrew.cmu.edu/system/archive/cmu/depot**.

Any questions about `depot` can be sent to depot+@andrew.cmu.edu. `Depot` does not require `AFS`.

References

[Ande91] Anderson, Paul. "Managing Program Binaries In a Heterogeneous UNIX Network." *LISA V Proceedings.* 1991. pp. 1-9.

[Coly92] Colyer, Wallace; Held, Mark; Markley, David, and Wong, Walter. "Software Management in the Andrew System." *AFS User's Group Proceedings.* June 1992.

[Held92] Held, Mark, and Neuhart, Dawn. *Software Management in the Andrew Distributed* UNIX *System at CMU.* Computing Services, Carnegie Mellon University. 1992.

[Manh90] Manheimer, Kenneth, Warsaw, Barry, Clark Stephen, and Rowe, Walter. "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries." *LISA IV Proceedings.* 1990. pp. 37-46.

[Saty85] Satyanarayanan, M.; Howard, J. H; Nichols, D. A.; Sidebotham N., and Spector A. Z. "The ITC Distributed File System: Principles and Design." *Proceedings of the 10th ACM Symposium on Operating System Principles*. 1985.

[Sell91] Sellens, John. "Software Maintenance in a Campus Environment: The Xhier Approach." *LISA V Proceedings.* 1991. pp. 21-44.

[Shaf89] Shafer, Stephen, and Thompson, Mary. *The SUP Software Upgrade Protocol.* Carnegie Mellon University, School of Computer Science. 1988. Available from mach.cs.cmu.edu in **/usr/mach/public/doc/sup.ps**.

[Side86] Sidebotham, R. N. "Volumes: The Andrew File System Data Structuring Primitive." *Technical Report CMU-ITC-053.* Information Technology Center, Carnegie Mellon University. 1986.

[Wall91] Wall, Larry, and Schwartz, Randal L. *Programming perl.* O'Reilly and Associates, Inc. 1991.

[Youn85] Yount, Russell. *Package.* Academic Services. Carnegie Mellon University. 1985.

Author Information

Wallace Colyer is the Andrew Systems Manager at Carnegie Mellon University. He began as a User Consultant specializing in workstation administration issues. Time and a variety of departmental reorganizations found him in charge of the entire system. Send Email to wally+@cmu.edu.

Walter Wong obtained a B.S. in Cognitive Science at Carnegie Mellon University in 1991. By that time, however, he was already involved with system administration issues in a distributed computing environment. Rather than basking in the glory of a fine graduate school in a small college town, Walter stayed at Carnegie Mellon to be a system administrator and programmer for the Andrew Systems Group. Send Email to Walter.C.Wong@cmu.edu.

Both authors may be reached via the postal system at:

Computing Services
Carnegie Mellon University
4910 Forbes Avenue
Pittsburgh, PA 15213-3891

Examples

The following example is a simple use of `depot` to integrate two collections into an environment called **/usr/test**. A directory called **depot** is created under **/usr/test** which houses the configuration files and collections. There are two collections *col1* and *col2*. Each has its own directory hierarchy which is shown below.

```
/usr/test/depot/col1
      bin/prog1
      bin/prog2
      man/man1/prog1.1
      lib/libprog1.a

/usr/test/depot/col2
      bin/prog3
      man/man1/prog3.1
      lib/libprog3.a
```

A simple configuration file is created which tells `depot` to use the modification times to see if a file has changed.

```
% cat /usr/test/depot/custom.depot
usemodtimes: true
```

Running `depot` will integrate these two collection, *col1* and *col2,* with a common **man**, **lib**, and **bin** directory.

```
% cd /usr/test/depot
% depot -B [this builds the initial database]
% depot -va [-v makes depot verbose; -a updates all collections]
DIRECTORY ..
MKDIR ../man
MKDIR ../man/man1
LINK depot/col2/man/man1/prog3.1 ../man/man1/prog3.1
LINK depot/col1/man/man1/prog1.1 ../man/man1/prog1.1
MKDIR ../lib
LINK depot/col2/lib/libprog3.a ../lib/libprog3.a
LINK depot/col1/lib/libprog1.a ../lib/libprog1.a
MKDIR ../bin
LINK depot/col2/bin/prog3 ../bin/prog3
LINK depot/col1/bin/prog2 ../bin/prog2
LINK depot/col1/bin/prog1 ../bin/prog1
Backing up old database .. done
Moving in new database .. done
```

The following is the directory hierarchy, reflecting the union of *col1* and *col2*, created under **/usr/test**.

```
/usr/test
      bin
            prog1
            prog2
            prog3
      lib
            libprog1.a
            libprog3.a

      man/man1
            prog1.1
            prog3.1
```

By adding the mapcommand line to the configuration file, the actual files are copied out of the collection and into the **/usr/test** hierarchy.

```
% cat custom.depot
usemodtimes: true
```

```
*.mapcommand: copy
% depot -va
DIRECTORY ..
DIRECTORY ../man
DIRECTORY ../man/man1
REMOVE ../man/man1/prog3.1
COPY depot/col2/man/man1/prog3.1 ../man/man1/prog3.1.NEW
RENAME ../man/man1/prog3.1.NEW ../man/man1/prog3.1
UTIMES ../man/man1/prog3.1 Wed Aug 19 10:52:11 1992
REMOVE ../man/man1/prog1.1
COPY depot/col1/man/man1/prog1.1 ../man/man1/prog1.1.NEW
RENAME ../man/man1/prog1.1.NEW ../man/man1/prog1.1
UTIMES ../man/man1/prog1.1 Wed Aug 19 10:51:19 1992
DIRECTORY ../bin
REMOVE ../bin/prog3
COPY depot/col2/bin/prog3 ../bin/prog3.NEW
RENAME ../bin/prog3.NEW ../bin/prog3
UTIMES ../bin/prog3 Wed Aug 19 10:51:56 1992
REMOVE ../bin/prog1
COPY depot/col1/bin/prog1 ../bin/prog1.NEW
RENAME ../bin/prog1.NEW ../bin/prog1
UTIMES ../bin/prog1 Wed Aug 19 10:51:03 1992
REMOVE ../bin/prog2
COPY depot/col1/bin/prog2 ../bin/prog2.NEW
RENAME ../bin/prog2.NEW ../bin/prog2
UTIMES ../bin/prog2 Wed Aug 19 10:51:03 1992
DIRECTORY ../lib
REMOVE ../lib/libprog3.a
COPY depot/col2/lib/libprog3.a ../lib/libprog3.a.NEW
RENAME ../lib/libprog3.a.NEW ../lib/libprog3.a
UTIMES ../lib/libprog3.a Wed Aug 19 10:52:21 1992
REMOVE ../lib/libprog1.a
COPY depot/col1/lib/libprog1.a ../lib/libprog1.a.NEW
RENAME ../lib/libprog1.a.NEW ../lib/libprog1.a
UTIMES ../lib/libprog1.a Wed Aug 19 10:51:29 1992
Backing up old database .. done
Moving in new database .. done
```

If a new file is added to a collection, it will be integrated into the environment by running `depot` again. Thus, the file **/usr/test/depot/col2/bin/prog4** is added to the collection *col2*.

```
% depot -va
DIRECTORY ..
DIRECTORY ../bin
COPY depot/col2/bin/prog4 ../bin/prog4.NEW
RENAME ../bin/prog4.NEW ../bin/prog4
UTIMES ../bin/prog4 Wed Aug 19 10:59:10 1992
Backing up old database .. done
Moving in new database .. done
```

A new environment, **/usr/test2**, can be created that builds upon the collections in the **/usr/test** environment. Under the **depot** directory in **/usr/test2** we have a newer versions of *col2* and a new collection called *col3*.

```
/usr/test2/depot/col2
      bin/prog3
      bin/prog4
      man/man1/prog3.1
      lib/libprog3.a

/usr/test2/depot/col3
      bin/prog5
      lib/prog5/fonts/prog5.font
```

```
% cd /usr/test2/depot
% cat custom.depot
usemodtimes: true
*.searchpath: /usr/test2/depot,/usr/test/depot
% depot -B
% depot -va
DIRECTORY ..
MKDIR ../man
MKDIR ../man/man1
LINK /usr/test2/depot/col2/man/man1/prog3.1 ../man/man1/prog3.1
LINK /usr/test/depot/col1/man/man1/prog1.1 ../man/man1/prog1.1
MKDIR ../lib
LINK /usr/test2/depot/col2/lib/libprog3.a ../lib/libprog3.a
LINK /usr/test/depot/col1/lib/libprog1.a ../lib/libprog1.a
MKDIR ../bin
LINK /usr/test2/depot/col3/bin/prog5 ../bin/prog5
LINK /usr/test/depot/col1/bin/prog2 ../bin/prog2
LINK /usr/test/depot/col1/bin/prog1 ../bin/prog1
LINK /usr/test2/depot/col2/bin/prog4 ../bin/prog4
LINK /usr/test2/depot/col2/bin/prog3 ../bin/prog3
LINK /usr/test2/depot/lib/prog5 ../prog5
Backing up old database .. done
Moving in new database .. done
```

This creates a new environment, *test2*, by using *col1* from the *test* environment along with the newer version of *col2* and a new collection, *col3*, from the *test2* environment. For **lib/prog5**, link optimization was accomplished. Since the only collection installing into the **lib/prog5** directory was *col3* and the entire directory was being imported, the symbolic link was made at the highest point in the tree.