Depot: A Tool for Managing Software Environments

Wallace Colyer & Walter Wong
Computing & Communications
Carnegie Mellon University

April 10, 1992

Abstract

Depot is a software management tool providing 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 the collections. This common directory is defined by *depot* as the software environment. Interactions and intersections between collections are handled through a set of configuration options.

Depot facilitates the introduction, update and removal of collections in a software environment. Custom environments and complete test environments can be created for individual machines or sets of machines. Collections with unexpected problems can be replaced with previous versions or simply removed. Individual collections can transparently be moved from remote filesystems to the local disk of workstations. 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 the lack of adequate 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 software providers make of the installation environment.

To properly maintain 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, both independently and in the destination environments. If a critical problem escapes the testing process, the software environment must be quickly and easily restorable to a previous working state.

The emergence and popularity of distributed computing has compounded this problem. A large heterogeneous environment of thousands of workstations and hundreds of software package aggravates the already existing problems and adds some new problems which must be overcome. There becomes a need to distribute and install software on remote machines, with different architectures, customizations, and configurations. Any procedures must also minimize the workload of the system administrators.

Many solutions that can manage a distributed software environment often bring back the load and availability problems of timesharing systems by increasing the dependence on centrally maintained services. To prevent this local workstations should be able to locally cache commonly used files as well as maintain a core set of functionality in light 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 create an environment which can address all these issues. These objects can then be layered to create the larger user visible environment. Thus, the environment can be looked at either as a whole or in parts. Large complex environments can be created utilizing the work done by a variety of different people who potentially are unaware of the other work being done.

Motivation

In the past software had been installed directly into the `/usr/local` tree. As the number of applications increased, the maintenance process became increasingly difficult. No records were kept of the files installed with a software package. This resulted in outdated files from previous versions of software wasting disk space. Similarly, when two separate applications had files of the same name installed into the same directory they would overwrite the conflicting file during the installation process and the resulting environment would be seemingly random. It was apparent that there was a serious problem when the software manager would make posts containing lists of hundreds of files asking if anyone knew if they still belonged in `/usr/local`.

We felt that there were four key components 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 all its files. This helps tremendously in finding and reporting problems as well as cleaning up the environment when updates occur or the software becomes obsolete. Furthermore, a complete software package can be distributed or shared by simply specifying the location of the collection.

The Depot [Manh90] developed at the National Institute of Standards and Technology 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 potentially very long list of items on his path. We considered this approach to be too cumbersome in our environment as many problems had previously been encountered making changes to the search paths of the over 10,000 users of our system, many of whom use our filesystem from decentralized departmental computing facilities. This solution also doesn't fully take into consideration the problem of two applications installing binaries with the same name. Path conflicts and ordering would be a constant problem.

Integration

It is necessary to not only maintain independent collections for the sake of administrative convenience, but it is also important to unify the collections into a single directory hierarchy. This eases the task of understanding the environment for the users and deals with the need for certain applications to share common directories.

Many applications need to share directories where common files are kept. Often index files are 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.

Though the collections are kept physically separated for administrative convenience, they should later be integrated into a common directory hierarchy. This not only solves the problems of path ordering and forces the conflicts to be resolved, but allows collections to later be broken up into smaller units without needing to perform major structural modifications.

Mobility

The most obvious reason for being able to quickly and easily 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 truly be tested by creating a duplicate of destination environment. For example, it should be possible to generate a software environment that is identical to the current released environment with the sole exception of the collection to be tested. Also, if a major problem were uncovered it should be possible to restore the old environment quickly and simply.

Distributed filesystems are ideal for utilizing 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 in a manner as transparent as possible. In this manner, rarely used applications may be stored on the remote filesystem, thereby conserving local disk space, while 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 has really addressed the issue of mobility. Most packages provide only one environment (`/software` in the case of *Xhier* and `/depot` in the case of NIST's *The Depot*). Ideally, multiple environments should be possible. For example, it is desirable to have an environment for fully supported software, such as `/usr/local` and another environment for "unsupported but useful" software, such as `/usr/contributed`. Moreover, it should be possible for software to easily move collections from `/usr/contributed` to `/usr/local` and vice versa.

Simplicity

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

Alternatively, simplicity and understandability can be maintained by having each collection imported into the software environment using its own internal directory hierarchy. A file placed in the `bin` directory of a collection should appear in the `bin` directory of the environment. All that the software installer should need to determine is the desired directory hierarchy and when it 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*, when two collections try to install the same file in the same place.

Additionally, the system can be kept simple by leaving distribution to an alternate mechanism. For example a distributed filesystem, such as *AFS*[1] [Saty85], or standard software distribution tools such as *rdist* and *SUP* [Shaf88] may be used. It may be important to incorporate distribution for certain classes of machines such as laptops and other computers connected by slow or unreliable network connections, but any distribution solution should be optional.

Taking all three of these issues into consideration a system which requires relatively little work can be setup to maintain multiple environments, to allow software to be quickly and easily installed and backed out, and to allow individual workstations to be customized by installing different software and by how the software is installed on the disk. All this can be done while minimizing the overall complexity of maintaining large software environments.

1. The Andrew Filesystem (AFS) is a large scalable filesystem and is available from the Transarc Corporation.

Implementation

Depot is a tool which manages software environments by integrating multiple independent collections into a single directory hierarchy and achieves the goals of independence, integration, mobility and simplicity.

With each invocation *depot* processes an independent software environment. The software environment starts with a specified directory hierarchy and includes everything within it, including subdirectories [see figure 1]. One common software environment is `/usr/local`. The concept of a software environment can also be extended to include the entire workstation.

Depot defines the environment as the union of a set of software collections [see figure 2]. *Depot* creates the environment by integrating the structure of each collection into a single directory hierarchy. *Depot* will fail prior to updating to performing any updates if there are any conflicts between 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. Conflicts between collections can be resolved either by specifying that one collection overrides another or by using configuration options to move files or directory hierarchies.

To incorporate collections into an environment, specific collections and the path to their location can be listed, search paths can be provided where the first instance of

Figure 1: A software environment

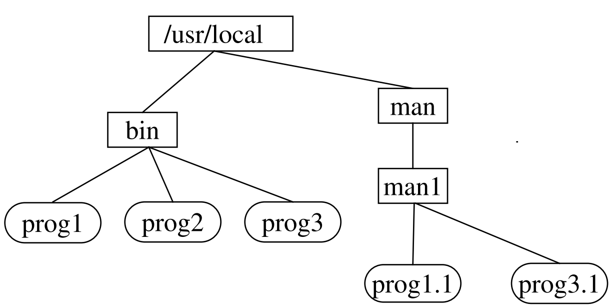
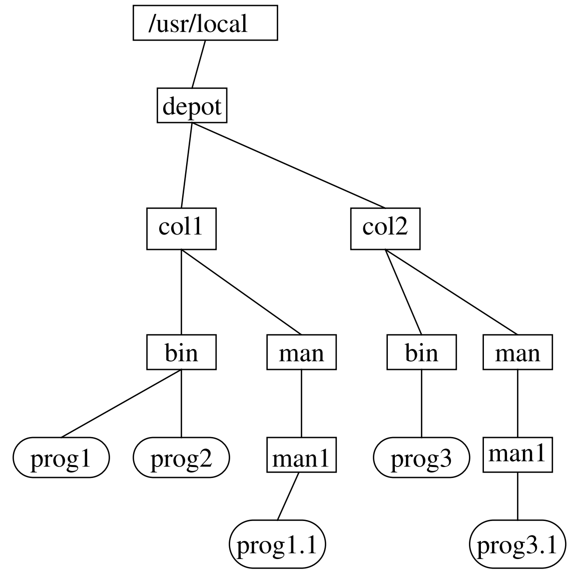


Figure 2: The collections



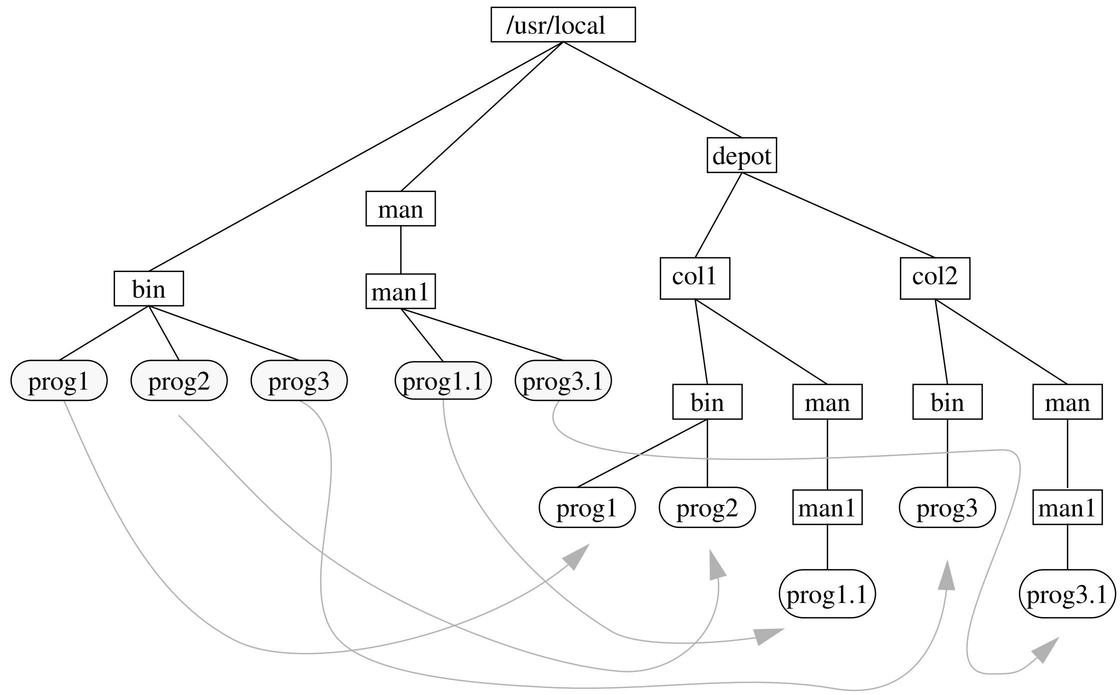
each collection within the path will be used, or a combination of both methods can be used. If the use of version numbers is turned on *depot* will choose the highest version of the first instance of the collection on the search path.

There are currently two ways a collection can be integrated, by copying or by 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, the links are made at directory level wherever possible. With the other option, copy, every file and directory is copied into the target environment. Symbolic links in the collections are copied exactly to the target environment [see figure 3].

In many cases multiple collections need to install files into a common directory. Often index files are kept of the contents of the directory, such as `fonts.dir` in the case of *X11* font directories. Through configuration options there is an automated way of updating the index files. A command can be run whenever a collection or directory has changed.

Before integrating the collections into one environment

Figure 3: The environment made with symbolic links



depot verifies that the environment is consistent. There must be no conflicts and no files which do not belong to a collection or are not marked as special files in the configuration files must be in the environment. Any conflicts found will result in *depot* exiting and stray files will be deleted, unless otherwise specified. Then *depot* will check to see if any of the collections on its paths have changed, have been added or deleted. All the new collections will added to the environments, all the removed collections will be deleted, and any necessary changes for modified collections will be made. If the files are copied into the environment any time a file changes in the collection specific directory it will be copied to the environment.

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 both administrators and software developers to understand and use, achieving the goal of simplicity.

Limitations

Depot is only able to operate inside a single environment. 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 or for the environment depot maintained to be extended to the entire workstation.

In previous version, depot was only able to operate at the collection level. For example, there was no way to specify individual files or directories to be copied, either the entire collections was copied or the entire collection was linked. This strict separation into collections with no single file operations has proven to be too strict. Target specific options have been added to permit individual files or directories to be copied, linked, deleted or ignored, regardless of their collection of origin. For example, several collections may install fonts into the `lib/X11/fonts` directory and the environment maintainer may wish them always copied regardless of the collection they come from. Similarly, 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 does not modify the resulting structure the sanctity of the collection is maintained and a great deal of flexibility is achieved for the environment maintainer.

Currently *depot* is somewhat inefficient at dealing with very large environments. The time it takes to search its databases and to `stat(2)` source directories for changed collections increases as the system grows. Some enhancements have been made to speed up the checking of collections within *AFS* volumes,[2] 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.

By making *depot* so simple, some additional tools are required for distribution and detailed tracking the actual environments. Mark Held [Held92] describes some of the other tools being used with *depot* to manage the Andrew software environment.

It has recently become obvious that *depot* could easily replace `package` [Youn85], the tool we currently use, and become a complete workstation manager. In order for this to be possible, many of these issues must be addressed. *Depot* must also be capable of dealing with target specific mode changes. The environment maintainer may wish a file installed with a different set of modes than it was installed in the collection from which it originated. 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.

Conclusion

In the Andrew environment, where *depot* was developed, it would be unthinkable to return to days before 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 more man hours were put into maintaining the environment than today, 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 and 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 three factors discussed earlier. The concept of combining independence and integration provided the manageability we needed without sacrificing consistency that users demand. Mobility gives us flexibility in configuration and testing. Finally, the simplicity has made it popular with the 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. The current version of *depot* was written by Sohan C. Ramakrishna-Pillai who did an excellent job fighting off creeping featurism.

2. Volumes are containers of Unix filesystems very similar to partitions. They are the administrative unit of *AFS*.[Side86]

A very talented group is currently working on the current and future versions of *depot*. They are specifically discussing the future of *depot* as a total workstation manager. This group consists of Sohan C. Ramakrishna-Pillai, Mark Held, Wallace Colyer, Walter Wong, John Myers, Dan Lovinger, and Chris Newman.

A final set of thanks goes to Mark Held, who is leading the Herculean 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*.

AFS is a trademark of the Transarc Corporation.

Unix is a trademark of AT&T.

Availability

Depot is available via anonymous ftp from:

`export.acs.cmu.edu [128.2.35.66]`
`/pub/depot`

Any questions about *depot* can be sent to `depot+@andrew.cmu.edu.`

References

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

[Held92] Held, Mark. *Software Management in the Andrew Distributed UNIX System at CMU.* Internal documentation, Computing & Communications, Carnegie Mellon University. 1992. This document is available through *AFS* in the directory `/afs/andrew.cmu.edu/acs/asg/doc/software`.

[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. This document is available through anonymous ftp at `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*. Internal obsolete documentation. Academic Services. Carnegie Mellon University. 1985.

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 telling *depot* to use the modification times to tell if a file has changed.

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

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

```
% cd /usr/test/depot
% depot -B [this builds the initial database]
% depot -va
DIRECTORY ..
MKDIR ../lib
LINK depot/col2/lib/libprog3.a ../lib/libprog3.a
LINK depot/col1/lib/libprog1.a ../lib/libprog1.a
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 ../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
```

Now a directory hierarchy reflecting the union of `col1` and `col2` has been created under `/usr/test`.

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

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

By simply adding a line to the configuration file we can copy the actual files out of the collection into the `/usr/test` hierarchy.

```
% cat custom.depot
usemodtimes: true
*.mapcommand: copy
% depot -va
DIRECTORY ..
DIRECTORY ../lib
REMOVE ../lib/libprog3.a
COPY depot/col2/lib/libprog3.a ../lib/libprog3.a
UTIMES ../lib/libprog3.a Sat Apr 4 13:34:50 1992
REMOVE ../lib/libprog1.a
COPY depot/col1/lib/libprog1.a ../lib/libprog1.a
UTIMES ../lib/libprog1.a Sat Apr 4 13:34:36 1992
DIRECTORY ../bin
REMOVE ../bin/prog3
COPY depot/col2/bin/prog3 ../bin/prog3
UTIMES ../bin/prog3 Sat Apr 4 11:23:16 1992
REMOVE ../bin/prog1
COPY depot/col1/bin/prog1 ../bin/prog1
UTIMES ../bin/prog1 Sat Apr 4 11:22:25 1992
REMOVE ../bin/prog2
COPY depot/col1/bin/prog2 ../bin/prog2
UTIMES ../bin/prog2 Sat Apr 4 11:22:28 1992
DIRECTORY ../man
DIRECTORY ../man/man1
REMOVE ../man/man1/prog3.1
COPY depot/col2/man/man1/prog3.1 ../man/man1/prog3.1
UTIMES ../man/man1/prog3.1 Sat Apr 4 13:35:10 1992
REMOVE ../man/man1/prog1.1
COPY depot/col1/man/man1/prog1.1 ../man/man1/prog1.1
UTIMES ../man/man1/prog1.1 Sat Apr 4 11:22:43 1992
Backing up old database .. done
Moving in new database .. done
```

If we add a new file to a collection, it will be integrated by simply running *depot* again. The file `/usr/test/depot/col2/bin/prog4` is added to the collection `col2`.

```
% depot -av
DIRECTORY ..
DIRECTORY ../lib
DIRECTORY ../man
DIRECTORY ../man/man1
DIRECTORY ../bin
COPY depot/col2/bin/prog4 ../bin/prog4
UTIMES ../bin/prog4 Sat Apr 4 13:45:52 1992
Backing up old database .. done
Moving in new database .. done
```

We could setup a new environment `/usr/test2` which 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/test/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 -av
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/col3/lib/prog5 ../lib/prog5
LINK /usr/test/depot/col1/lib/libprog1.a ../lib/libprog1.a
LINK /usr/test2/depot/col2/lib/libprog3.a ../lib/libprog3.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
Backing up old database .. done
Moving in new database .. done
```

This created a new environment, `test2`, using `col1` from the `test` environment along with the newer version of `col2` and a new collections `col3` from the `test2` environment. Note that 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 possible point in the tree.