Safe and Policy-Free Software Deployment with Nix

Eelco Dolstra and Eelco Visser
Utrecht University

Merijn de Jonge
Eindhoven University of Technology

Abstract

Existing systems for software deployment are neither safe nor sufficiently flexible. Primary safety issues are the inability to enforce reliable component dependency specification, and a lack of support for multiple versions or variants of a component. This renders deployment operations such as upgrading or deleting components dangerous and unpredictable. A deployment system must also be flexible (i.e., policy-free) enough to support both centralised and local package management, and to allow a variety of mechanisms for transferring components. In this paper we present Nix, a deployment system that addresses these issues through a simple technique of using cryptographic hashes to compute unique paths for component instances.

1 Introduction

Software deployment is the act of transferring software to the environment where it is to be used. This is a deceivingly hard problem: a number of requirements make effective software deployment difficult in practice, as most current systems fail to be sufficiently *safe* and *flexible*.

The main safety issue that a software deployment system must address is *consistency*: no deployment action should bring the system into an inconsistent state. For instance, an installed component should never be able to refer to any component not present in the system; and upgrading or removing components should not break other components or running programs [13]. It must also be possible to have overlapping packages installed and operational at the same time (such as multiple versions and variants of the same component). No duplicate components should be installed: if two components have a shared dependency, that dependency should be stored exactly once.

Deployment systems must be flexible. They should support both *centralised* and *local package management*: it should be possible for both site administrators and local users to install applications, for instance, to be able to use different versions and variants of components. Finally, it must not be difficult to support deployment both in source and binary form, or to define a variety of mechanisms for transferring components. In other words, a deployment system should provide flexible *mechanisms*, not rigid *policies*.

Despite much research in this area, proper solutions have not yet been found. For instance, a summary of twelve years of research in this field indicates, amongst others, that many existing tools ignore the problem of overlapping packages and that end-user customisation has only been slightly examined [8]. Consequently, there are still many hard outstanding deployment problems (see Section 2), and there seems to be no general deployment system available that satisfies all the above requirements. Most existing tools only consider a small subset of these requirements and ignore the others.

In this paper we present Nix, a safe and flexible deployment system providing mechanisms that can be used to define a great variety of deployment policies. The prime features of Nix are: i) concurrent installation of multiple versions and variants; ii) atomic upgrades and downgrades; iii) multiple user environments; iv) safe dependencies; v) complete deployment; vi) transparent source and binary deployment; vii) safe garbage collection; viii) multi-level package management (i.e. different levels of centralised and local package management); ix) portability. These features follow from the fairly simple technique of using cryptographic hashes to compute unique paths for component instances.

The paper is structured as follows. Section 2 motivates the need for advanced software deployment technology. We give a high-level overview of Nix in Section 3 and describe its implementation in Section 4. We address policies for deployment in Section 5, and for user environment management in Section 6. In Section 7 we discuss related work. Conclusions and future work are discussed in Section 8.

2 Motivation

In this section we take a close look at the issues that a system for software deployment must be able to deal with.

Dependencies For safe software deployment, it is essential that the *dependencies* of a component are correctly identified. For correct deployment of a component, it is necessary not only to install the component itself, but also

all components which it may need. If the identification of dependencies is incomplete, then the component may or may not work, depending on whether the omitted dependencies are already present on the target system. In this case, deployment is said to be *incomplete*.

As a running example for this paper we will use the *Subversion* version management system [6]. It has several (optional) dependencies, such as on the Berkeley DB database library. When we package the Subversion component for deployment, we must take this into account and ensure that the Berkeley DB component is also present on each target system. But it is easy to forget this! This is because such dependencies are often picked up "silently". For instance, Subversion's configure script will detect and use Berkeley DB automatically if present on the build system. If it is present on the target system, Subversion will happen to work; but if it is not, it won't: an incomplete deployment. Some existing deployment systems use various tricks to automate dependency identification. E.g., RPM [12] can use the ldd tool to scan for shared library dependencies. However, such approaches are either not general enough or not portable.

Variability Components may exist in many *variants*. Variants occur when different versions exist (i.e., almost always), and when a component has optional features that can be selected at build time. This is known as variability [21]. The Subversion component has several optional features, such as: whether the built component shall support OpenSSL encryption and authentication; whether only a Subversion client shall be built; whether an Apache server module shall be built so that Subversion can act as a WebDAV server; and whether debug support shall be enabled. Of course, there also exist many different versions of Subversion, which we sometimes want to use in parallel (for instance, to test a new version before promoting it to production use on a server). A flexible deployment system should support the presence of multiple variants of a component on the same system. For instance, on a multi-user system different users may have different requirements and therefore need different variants; on a server system we may want to test a new component before upgrading critical server software to use it; or other components may have conflicting requirements on some component.

Consistency Unfortunately, most package management disciplines do not support variants very well. Deployment operations (such as installing, upgrading, or renaming a component) are typically *destructive*: files are copied to certain locations within the file system, possibly overwriting what was already there. This can destroy the consistency among components: if we upgrade or delete some component, then another component that depends on it may cease to work properly. Also, it makes it hard to have multiple variants of a component installed concurrently, that is, different versions of the component, or a version built with different parameters. For instance, the RPM packages for Subversion install files such as /usr/bin/svn, making it impossible to have two versions installed at the same time. Worse, we might encounter unsatisfiable requirements, e.g., if two applications both require mutually incompatible versions of some library.

Atomicity Component upgrades in conventional systems are not *atomic*. That is, while a component is being overwritten with a newer version, the component is in an inconsistent state and may well not work correctly. This lack of atomicity extends beyond the level of individual components. When upgrading an entire system, for instance, it may be necessary to upgrade shared components such as shared libraries first. If these are not backwards compatible, then there will be a timing window in which components that use these fail to work properly.

Identification Third, variants make identification of dependencies surprisingly hard. We may say that a component depends on glibc-2.3.2, but what are the exact semantics of such a statement? For instance, it does not identify the build parameters with which glibc has been built, nor is there any guarantee that the identifier glibc-2.3.2 always refers to the same entity in all circumstances. Indeed, versions of Red Hat Linux and SuSE Linux both have RPM packages called glibc-2.3.2, but these are not the same, not even at the source level (they have vendor-specific patches applied).

Source/binary deployment We must often create both "source" and "binary" packages for a component. Creating the latter is unfortunate, since binary deployment can be considered an optimisation of source deployment, i.e., it uses fewer resources on the target system. Ideally, this would happen automatically and transparently, but in practice, the creation and dissemination of binary packages requires explicit effort. This is particularly the case if multiple variants are required (which variants do we build, and how do users select them?).

The source/binary dichotomy complicates dependency specification, since a component can have different dependencies at build time and at run time that must be carefully identified. This is tricky, since a build time dependency can become a run time dependency if the construction process *stores* a reference to its dependencies in the build result—a *retained dependency*. For instance, various libraries such as OpenSSL are inputs to the Subversion build process. If these are shared libraries, then their full paths (e.g., /usr/lib/libssl.so.0.9.6) will be stored in the resulting Subversion executables. Thus, these build time dependencies become run time dependencies. However, if they are *statically* linked (which is a build time option of Subversion), then this does not occur. Thus, there is a subtle interaction between variant selection and dependencies.

Centralised vs. local package management To make software deployment efficient, system administrators should not have to install each and every application separately on every computer on a network. Rather, software installation should be managed centrally. On the other hand, computers or individual users may have individual software requirements. This requires local package management. Software deployment should cater for both local and centralised package management. It should not be hard to define site-local policies.

Storage efficiency Deployment would be simpler if we made all compositions as "static" as possible. For instance, if we only use statically as opposed to dynamically linked libraries, we can greatly reduce the number of run time dependencies. However, this is inefficient with respect to storage space. In the worst case, if we have n applications with m common dependencies, then this approach would yield space requirements of $\Theta(nm)$, instead of $\Theta(n + m)$ in the shared case.

3 Overview

The Nix software deployment system is designed to overcome the problems of deployment described in the previous section. The main ingredients of the Nix system are the *Nix store* for storing isolated installations of components[1]; *user environments*, providing a user view of a selection of components in the store; *Nix expressions*, specifying the construction of a component from its sources; and *build caches* for collecting and distributing already built components. These ingredients provide *mechanisms* for implementing a wide variety of deployment *policies*. In this section we give an overview of these ingredients from the perspective of users of the system. In the next section their implementation is described.

3.1 Nix Store

The fundamental problem of current approaches to software deployment is the confusion of *user space* and *installation space*. An end-user activates applications installed on a computer through a certain interface. This may be the *start menu* on Windows and other desktop environment, or the PATH environment variable in command-line interfaces on Unix-like systems. These interfaces form *user space*. Deployment is concerned with making applications available through such interfaces by installing all files necessary for their operation in the file system, i.e., in *installation space*.

[1]The name *Nix* is derived from the Dutch word *niks*, meaning *nothing*; build actions do not see anything that has not been explicitly declared as an input.

Mainly due to historical reasons—deployment was often done manually—user space and installation space are commonly identified. For instance, to keep the list of directories in the PATH manageable, applications are installed in a few fixed locations such as /usr/bin. Thus, management of the end-user interface to applications is equal to physical manipulation of installation space, entailing all the problems discussed in the previous section.

In Nix, user space and installation space are separated. User space is a *view* of installation space. Applications and all programs and libraries used to implement them are installed in the Nix *store*. Each component is installed in a separate directory in the store. Directory names in the store are chosen so as to uniquely identify revisions and variants of components. This identification scheme goes beyond simple name+version schemes, since these cannot cope with variants of the same version of a component. Thus, multiple versions of a component can coexist in the store without interference. Furthermore, due to this unique identification an installed component can be shared by all components that depend on it.

3.2 User Environments

A Nix *user environment* consists of a selection of applications from the store currently relevant to a user. (A "user" can be a human user, but also system users such as daemons and servers, which may need a specific selection to be visible.) This selection may be implemented in various ways, depending on the interface used by the user. In the case of the PATH interface, a user environment is implemented as a single directory—the counterpart of /usr/bin—containing symbolic links (or wrapper scripts on systems that don't support them) to the selected applications. Thus, manipulation of the user environment consists of manipulation of this collection of symbolic links, rather than directories in the store. Installation of an application in user space entails adding a symbolic link to a file in the store and uninstallation entails removing this symbolic link instead of physically removing the corresponding file from the file system.

While other approaches also use a directory with symbolic links, these are composed manually and/or are only provided in a single location. In Nix an environment is a component in the store. Thus, any number of environments can coexist and variant environments can be composed with tools (Figure 1). This separation of user space and installation space allows the realization of many different deployment scenarios. The following are some typical examples:

- A user environment may be prescribed by a system administrator, or may be adapted by individual users.
- Different users on the same system can compose dif-

nix-env -p *profile*
: Switch to another user environment, i.e., make *profile* the current user environment.

nix-env -I *system*.nix
: Import component descriptions from a collection in file *system*.nix.

nix-env -q
: List components installed in the current user environment.

nix-env -qs
: List the *available* components and give their status, i.e., whether they are installed in the store and/or in the current user environment.

nix-env -i *component*
: Install a component in the current user environment; install it in the store if it is not installed there yet.

nix-env -e *component*
: Uninstall (erase) a component from the current user environment.

nix-env -u *component*
: Upgrade to a new version of a component, i.e., replace component with a new version *in the user environment*.

nix-store –gc
: Garbage collection: compute components in the store which are not needed in any user environment and delete them.

nix-pull *url*
: Pull description of pre-built components from a network cache.

nix-push *url component*
: Copy a component from the local store to a network cache.

Figure 1: Summary of high-level Nix commands

ferent user environments, or can share a common environment.

- A single user can maintain multiple 'profiles' for use in different working situations.
- A user can experiment with a new version of a component while keeping the old (stable) version around for regular tasks.
- Upgrading to a new version or rolling back to an old one is a matter of switching environments.
- Removal of unused applications can be achieved by automatic *garbage collection*, taking the applications in user environments as roots.

These scenarios can be realised using the high-level Nix commands summarised in Figure 1. The interface of the nix-env command is similar to that of rpm. The difference with rpm is that these commands do not directly initiate destructive updates of the file system, but of user environments only. Installation and deletion of components in the store is initiated transparently when necessary, and never disrupts existing environments.

3.3 Nix Expressions

Installation of components in the store is driven by *Nix expressions*. These are declarative specifications that describe all aspects of the construction of a component, i.e., obtaining the sources of the component, building it from those sources, the components on which it depends, and the constraints imposed on those dependencies. Rather than having specific built-in language constructs for these notions, the language of Nix expressions is a simple functional language for computing with *sets of attributes*. Figure 2 shows a Nix expression for generating variants of the Subversion system; it features most typical constructs of the language. Figure 3 shows an application of this function. We'll use these examples to explain the elements of the language.

Derivation The body of the expression is formed by an application of the primitive function derivation to an *attribute set* {key=value;...}. The set contains two attributes required by the derivation function: the builder attribute indicates the script to use to build the component, and the system attribute specified the target platform on which the build is to be performed. The other attributes define values for use in the build process and are passed to the build script as environment variables; these attributes have no special meaning to Nix. The name attribute is a symbolic identifier for use in the high-level user interface; it is not the unique identifier of the component.

Parameters In order to describe variants of a component, an expression can be *parameterised*, i.e., turned into a *function* from Nix expressions to Nix expressions. Thus, the Subversion expression is parameterised with expressions describing the components on which it depends (e.g., openssl, httpd, stdenv), options that select variations (e.g., clientOnly, sslSupport), and a utility (fetchurl). The stdenv component provides the all the basic tools that one would expect in a Unix-like environment, e.g., a C compiler, linker, and standard Unix utilities. Parameters are instantiated in a function application. For example, the expression in Figure 3 instantiates the Subversion expression by assigning values to several

```
{ clientOnly , apacheModule , sslSupport
, stdenv , fetchurl , openssl , httpd , db4 }:

assert expat ≠ null;
assert ¬clientOnly → db4 ≠ null;
assert apacheModule → ¬clientOnly;
assert sslSupport → (openssl ≠ null
   ∧ (apacheModule → httpd.openssl = openssl));

derivation {
   name = "subversion-0.32.1";
   system = stdenv.system;

   builder = ./builder.sh;
   src = fetchurl {
      url = http://svn.collab.net/.../subversion-0.32.1.tar.gz;
      md5 = "b06717a8ef50db4b5c4d380af00bd901";
   };

   clientOnly = clientOnly;
   apacheModule = apacheModule;
   sslSupport = sslSupport;

   stdenv = stdenv;
   openssl = if sslSupport then openssl else null;
   httpd = if apacheModule then httpd else null;
   db4 = if clientOnly then null else db4;
}
```

Figure 2: Nix expression for family of Subversion components

of its parameters. (The argument expressions are mostly variables bound in a context not shown here.)

The value of the src attribute is another example of functional computation. Its value is an application (function call) of fetchurl, which is a parameter of the Subversion function, to the parameter attribute specifying the URL and MD5 checksum of the tarball distribution of Subversion 0.32.1. The fetchurl expression provides a script for downloading the distribution and verifying it against the checksum.

Assertions In order to restrict the values that can be passed as parameters, a function can state assertions that should hold of the parameter values. For example, the expat library is always needed for Subversion, and the db4 database is needed when a local server is implemented. Also, *consistency* between components can be enforced. For instance, if both SSL and Apache support are enabled, then they Apache must link against the same OpenSSL library as Subversion, since at runtime, the Subversion code will be linked against Apache. If this were not enforce, link errors could ensue.

```
stdenv = import ...;
... (other component definitions)

subversion = (import subversion.nix) {
   clientOnly = false; apacheModule = false;
   sslSupport = true;
   stdenv = stdenv; fetchurl = fetchurl;
   openssl = openssl; httpd = apacheHttpd;
   db4 = db4; expat = expat;
};
```

Figure 3: A specific Subversion component

Build Script When a derivation is built, the build script pointed at by the builder attribute in the derivation is invoked. Environment variables are used to pass parameters. The builder interprets the environment variables to obtain the source distribution, unpack it, configure it, and invoke the appropriate build and installation actions. Any components on which the component depends are built by Nix, so the build script can assume that they are available. The source tree should be configured such that all files for the component are installed in the Nix store in the directory indicated by the special environment variable out, which is provided by Nix. Figure 4 shows the build script for Subversion. The largest part of the script is used to compute the configuration flags based on the options selected in the instantiation of the Nix expression. By using a user-definable script for implementing the build of a component, rather than building in a specific build sequence, no requirements have to be made on the build interface of source distributions.

3.4 Sharing Component Builds

The unique identification of a component in the store is based on all the inputs to the build process, thus capturing all special configurations of the particular variant being built. Thus, components can be identified exactly and deterministically. Consequently a component can be shared by all components that depend on it. Indeed we even get *maximal sharing*: if two components are the same, then they will occupy the same location in the store. This means that builds can be shared by users on the same machine.

Since the identification only depends on the inputs to the build process and the location of the store, store identifiers are even *globally unique*. That is, a component build can be safely installed in a Nix store on another machine. For this purpose, Nix provides support for maintaining a network cache of pre-built components. After building a component in the store it can be *pushed* to the cache. A user or administrator on another machine can *pull* (descriptions of) these components from the cache and use

```
buildinputs="$openssl $db4 $httpd $expat"
(Bring in GCC etc., and set up the environment.)
. $stdenv/setup

if ! test $clientOnly; then
    extraflags="–with-berkeley-db=$db4 $extraflags"
fi

if test $sslSupport; then
    extraflags="–with-ssl –with-libs=$openssl $extraflags"
fi

if test $apacheModule; then
    extraflags="–with-apxs=$httpd/bin/apxs ... $extraflags"
fi

tar xvfz $src || exit 1
cd subversion-* || exit 1
./configure –prefix=$out $extraflags || exit 1
make || exit 1
make install || exit 1
```

Figure 4: Build script for Subversion

them when possible to substitute for a complete build from source. Thus, deployment of binaries is achieved transparently, as an optimisation of a source-based deployment process.

3.5 Policies

Nix is *policy-free*. That is, the ingredients introduced above are *mechanisms* for implementing software deployment. A wide variety of *policies* can be based on these mechanisms.

For instance, depending on the type of organisation it may or it may not be desirable or possible that users install applications. In an organisation where homogeneity of workspaces is important, the selection and installation of applications can be restricted to system administration. This can be achieved by restricting all the operations on the store, and the composition of user environments to system administration. They may compose several prefab user environments for different classes of users. On the other hand, for instance in a research environment, where individual users have very specific needs, it is desirable that users are capable of installing and upgrading applications themselves. In this situation environment operations and the underlying store operations can be made available to ordinary users as well.

Similarly, Nix enables deployment at different levels of granularity, from a single machine, a cluster of machines in a local network, to a large number of machines on separate sites. By composing environments for all relevant pairs of machine and user requirements, a cache of prebuilt components can be filled. This cache can be used to automatically install components on a large number of machines.

Many other policies are possible; some are discussed in Sections 5 and 6.

4 Implementation

In this section we discuss the implementation of the Nix system. We provide an overview of the main components of the system, which we then discuss in detail.

4.1 The big picture

The main design goals of the Nix system are twofold: to *support concurrent variants*, and to *ensure complete dependency information*. It turns out that the solutions to these two problems are closely related. Behind these lies the meta-goal of *completeness*: the deployment process should transmit all dependencies necessary for the correct operation of a component. Other design goals are portability (we should not fundamentally rely on operating system specific features or extensions) and storage efficiency (identical components should not be stored more than once).

The first problem is dealing with variability, i.e., concurrent variants. As we hinted in the previous section, we support this by storing each variant of a component in a global *store*, where they have unique names and are isolated from each other. For instance, one version or variant of Subversion might be stored in /nix/store/eeeeaf42e56b-subversion-0.32.1[2], while another might end up in /nix/store/3c7c39a10ef3-subversion-0.34. To ensure uniqueness, these names are computed by hashing all inputs to the build process that built them.

The use of these names also provides a solution for the dependency problem. First, it prevents undeclared dependencies. While it is easy for hard-coded paths (such as /usr/bin/perl) to end up in component source, thereby causing a dependency that is easily forgotten while preparing for deployment, no developer would manually write down these paths in the source (indeed, being the hash of all build inputs, they are much too "fragile" to be included). Second, we can now actually *scan* for dependencies. For instance, if the string 3c7c39... appears in a component, we know that it has a dependency on a specific variant of Subversion 0.34. This in particular solves the problem of retained dependencies (discussed in Section 2): it is not necessary to declare explicitly those

[2]The actual names use 32 hexadecimal digits (from a 128-bit cryptographic hash), but they have been shortened here to preserve space.

build time dependencies that, through retention, become run time dependencies, since we can find them automatically.

With precise dependency information, we can achieve the goal of complete deployment. The idea is to always deploy *component closures*: if we deploy a component, then we must also deploy its dependencies, their dependencies, and so on. That is, we must always deploy a set of components that is closed under the "depends on" relation. Since closures are self-contained, they are the units of complete software deployment. After all, if a set of components is *not* closed, it is not safe to deploy, since executing one of the component might cause components to be referenced that are missing on the target system.

4.2 The store

Central to the Nix system is the *store*, which is a directory in the file system (typically /nix/store) where all components live, along with all information involved in building them:

- *Derivates* are the results of *derivations*, which are simply component build actions. A derivation must be automatic: no user intervention, other than initiation, should be involved. They must be *pure*: given the same inputs we should obtain the same derivate; we disregard "minor" impurity, such as the current time being stored in file time stamps.
- *Sources* are files not produced by Nix, but copied to the store to serve as input to derivations. For all intents and purposes, these are treated as derivates.
- *Store expressions* are auxiliary data used to describe derivations and closures. These are discussed in detail below.

As we stated above, each object in the store has a unique name, so that variants can co-exist. These names are called *store paths*. (Precisely how we compute these store paths is discussed in Section 4.4.) In Autoconf [1] terminology, each component has a unique prefix. The file systems content referenced by a store path is called a *store object*. Note that a given store path uniquely determines the store object. This is because two store objects can only differ if the inputs to the derivations that built them differ, in which case the store path would also differ due to the hashing scheme used to compute it. Also, a store object can never be changed after it has been built.

Figure 5 shows a number of derivates in the store. The tree structure simply denotes the directory hierarchy. The arrows denote dependencies, i.e., that the file at the start of the arrow contains the path name of the file at the end of the arrow. E.g., the program svn depends on the library libc.so.6, because it lists the file /nix/store/8d013ea878d0-glibc-2.3.2/lib/libc.so.6 as one of the shared libraries against which it links at runtime.

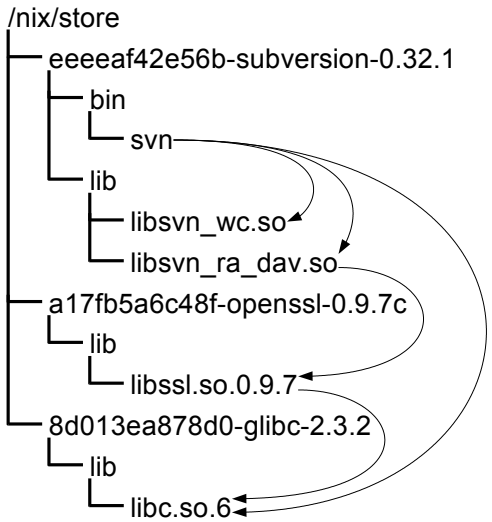

Figure 5: The store

Thus, components are stored in isolation from each other. This greatly simplifies certain aspects of package management. For instance, operations such as deleting a component, or asking to what component a certain file belongs, become trivial. E.g., for the former, we just recursively delete its directory in the store. This is *not* the case in more conventional file system layouts. If the Subversion component were stored in "global" locations such as /usr/bin/svn, /usr/lib/libsvn.so, and so on, then we must track to which component each file belongs. RPM, for instance, uses a database to accomplish this. However, this database (which essentially shadows the file system, since it provides meta-data for files) is critical; it cannot be easily recovered if it is destroyed, and it must be kept synchronised with the file system.

4.3 Store expressions

Building a component instance could be expressed directly in terms of Nix expressions, but there are several reasons why this is a bad idea. First, the language of Nix expressions is fairly high-level, and as the primary interface for developers, subject to evolution; i.e., the language changes to accommodate new features. However, this means that we would have to be able to deal with variability in the Nix expression language itself: several versions of the language would need to be able to co-exist in the store. Second, the richness of the language is nice for users but complicates the sorts of operations that we want to perform (e.g., building and deployment). Third, Nix expressions cannot easily be uniquely identified. Since Nix expressions can import other expressions scattered all

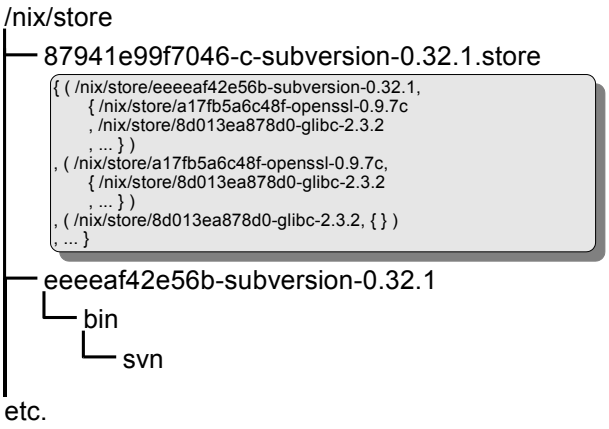

Figure 6: Closure value for Subversion

```
name = "subversion-0.32.1";
system = "i686-linux";
builder = <.../ccd431d728dd-c-builder.sh.store>;
src = <.../bd1189730f91-d-subversion-...tar.gz.store>;
clientOnly = false;
apacheModule = false;
sslSupport = true;
stdenv = <.../0acec22a0040-d-stdenv-nix-linux.store>;
openssl = <.../2557d3bd92cb-d-openssl-0.9.7c.store>;
httpd = null;
db4 = <.../8a3d549ed53f3-d-db4-4.0.14.nix.store>;
```

Figure 7: Subversion Nix expression after attribute evaluation

over the file system, it is not so straightforward to generate an identifier (such as a cryptographic hash) that uniquely identifies the expression. Finally, a monolithic architecture makes it hard to use different component specification formalisms on top of the Nix system (e.g., we might want to retarget Makefiles to Nix).

For these reasons Nix expressions are translated into the much simpler language of *store expressions*, just as compilers generally do the bulk of their work on simpler intermediate representations of the code being compiled, rather than on a full-blown language with all its complexities. There are only two types of values in the store expression language.

Closure values describe a closure in the store. These store the information that is essential for complete deployment. Above we defined a closure as a set of components that is closed under the "depends on" relation. We can now be more specific: it is a set of store paths, whose associated store objects do not refer to any store path not in this set. The method for computing closures is discussed in Section 4.5.

A closure value is a set of *closure elements*, which in turn are tuples consisting of a path in the store, along with all store paths referenced from the store object at this path. The latter paths must also occur in the closure, that is, as the left-hand side of some closure element.

Figure 6 shows a closure value expression for the store paths shown in Figure 5. Sets are denoted as $\{...\}$, and tuples as (..., ...). Note that the closure value expression encodes the dependencies indicated by the arrows in Figure 5, except that dependencies are described at the level of store paths, not at the level of individual files within store paths.

Derivation values encode all information required to perform a derivation. A derivation value can be *normalised* into a closure value by performing the build action (thus, store expressions form a very simple calculus with the normalisation operation as its only reduction rule). The result is a closure of the path created by the build action. A derivation value includes the following bits of information:

- Output path: the store path created by the derivation.
- Inputs: the paths of store expressions describing inputs to this derivation.
- Build platform: a characterising description of the system on which the build action is to be performed (e.g., i686-linux would be a minimal description).
- Builder: the path to an executable program that performs the build action. This path must be contained within one of the input closures.
- Command line arguments and environment variable bindings: these are used to communicate arbitrary parameters to the builder.

4.4 Translating Nix expressions to store expressions

Since we don't operate on Nix expression directly, but rather on store expressions, it is necessary to translate a Nix expression to one or more store expressions that (when normalised) result in the desired component(s). Below we discuss how this is achieved.

The Nix expression to be translated is first evaluated. Actual store expression creation is done as a side-effect of evaluating the primitive function derivation, which takes a set of attributes and evaluates each of them to construct a store derivation value. For instance, the derivation in Figure 3 has attributes name, system, openssl, and so on. Any attribute which is itself a derivation is added to the set of inputs of the derivation. Any attributes which refer to files (e.g., ./builder.sh) are copied to the store, using a name that is a hash of the file. Also, atomic closure values are created for these (that is, closures consisting of just the copied file just created in the store). The attribute

Figure 8: Derivation value for Subversion

system sets the build platform, while the attribute builder determines the builder. The attribute args sets the command line argument, and must evaluate to a list. All other attributes, in addition to determining the inputs closures, are also added as environment variable bindings. Figure 7 shows the Subversion derivation after evaluation of the attributes (for a specific selection of arguments). Attributes that evaluated to derivation or closure values are indicated between sharp brackets.

It is now necessary to determine the output path for the derivation (the store path where the result of the derivation is stored). The requirements here are that variants should have different store paths, and that identical components (i.e., not variants) should not be stored more than once. Note that selecting a cryptographically strong pseudo-random number satisfies the first requirement but not the second. So when are two components variants? This is when they differ with respect to *any* input to the build process that created them, i.e., sources, build parameters, dependencies, and the platform on which they were built. Observe that this implies that different versions are also variants. Thus, to ensure that two different components never clash, the paths in which they are stored must encode all build inputs in some way. An obvious solution is to use cryptographic hashes for this purpose, since any change to an input will cause the hash to change with very high probability.

With cryptographic hashes, we can compute the store path as follows. First, the output path of the derivation value being constructed is set to the Nix store, usually /nix/store. Then the derivation value is hashed cryptographically (Nix uses 128-bit MD5 hashes [18]). This hash is used to form the actual output path (e.g., /nix/store/eeeeaf42e56b-subversion-0.32.1). This scheme ensures that any change to the build inputs, as well as different store locations (e.g., /home/joe/nix/store instead of /nix/store), produces different hashes. A final point is the store path of the derivation itself. For this, we simply use the hash of the final derivation (e.g., /nix/store/0ba0329e59cb-d-subversion-0.32.1.store).

Figure 8 shows a few of the store expressions resulting from the translation of Figure 7. The top level derivation, for Subversion itself, contains all necessary bits of information: the output path, the intended build platform, the program to execute, command line arguments (none), environment variables, and the store paths of input store expressions.

4.5 Normalising store expressions

A derivation value is normalised into a closure value by performing the build action described by it. It is possible that a derivation has already been performed, in which case it should not be done again for performance reasons. Therefore, we maintain a persistent *successor* mapping that maps derivation values to their corresponding closure values, if known. Thus, if a successor for a derivation value is known, then we can normalise the successor (a closure value) instead.

If no successor is known, we must perform the build action. First, all inputs are normalised. (For inputs that are closure values, this is a no-op.) This ensures that all input closures are present. Then, the actual build action is performed by invoking the builder, with the command line arguments and environment values specified in the derivation value. Note that no other environment values are passed, e.g., PATH is unset. This prevents the builder from obtaining undeclared inputs, which would render the build action impure.

A build failure is signalled by a non-zero exit code from the builder. No successor is registered for a failed derivation. Thus, subsequent attempts to normalise the expression will simply retry the build. This is because, e.g., a disk full condition can cause a build to fail while it might succeed when repeated after this condition is removed.

If the builder succeeds, Nix verifies that the output path was in fact created. Nix then computes a closure value—the normal form of the derivation value—for the output paths. Here we need to address the problem of retained dependencies (Section 2). Since the output paths may contain references to paths in input closures, these ref-

erenced paths have to be included in the output closure as well. Formally, the paths to be included in a closure for some path p are

$$closure(p) = \{p\} \cup \bigcup_{p' \in refs(p)} closure(p')$$

where $refs(p)$ is the set of store paths referenced from the store object at p. Note that this is precisely the right-hand side of a closure element. Therefore, after building a path p, we only need to compute $refs(p)$ for that particular path, and not for the paths referenced by it, since we can obtain that information from their input closures.

The problem, then, is to discover $refs(p)$. We do this by scanning the full contents of path p (i.e., all file names, symlink targets, and contents of regular files in p) for the hash components of the input closures used to build them. E.g., if the path /nix/store/93f1...09aa is contained in an input closure, and we find the string 93f1...09aa in the output path, then we conclude that the output path has a reference to the former path and that therefore it should be included in the closure.

This approach is not fool-proof. It is possible for references to escape detection by encoding them in such a way that they become "invisible" to our scanner. For instance, since at present we just scan for these hash strings as 8-bit ASCII representations, references can be hidden by encoding them as UTF-16, by reversing them, or by compressing files. However, we have not yet encountered a single example of a reference being missed. Also, the scanning approach is not so strange as it might seem: it has been used to good effect for years in conservative garbage collectors [9].

Normalisation of the derivation expression shown in Figure 8 will yield the closure expression shown in Figure 6. Note that some inputs, such as the source code in /nix/store/b06717a8ef50-subversion-0.32.1.tar.gz, do not appear in the closure. This is because no reference to them appears in the output path. On the other hand, /nix/store/a17fb5a6c48f-openssl-0.9.7c *is* referenced (i.e., is a retained dependency), and so appears in the closure.

4.6 Substitutes

With just the mechanisms described above, Nix would be a source-based deployment system (like the FreeBSD Ports collection [2], or Gentoo Linux [3]), since all target systems would have to do a full build of all derivations involved in a component installation. This has the advantage of flexibility. Advanced users or system administrators can adapt Nix expressions to build a variant specifically tailored to their needs. For instance, required functionality disabled by default can be enabled, unneeded functionality can be disabled, or the components can be built with specific optimisation parameters for the target environment. The resulting derivates may be smaller, faster, easier to support (e.g., due to reduced functionality), and so on. On the other hand, the obvious disadvantages are that source-based deployment requires substantial resources on the target system, and that it is unsuitable for the deployment of closed-sourced products.

The Nix solution is to allow source-based deployment to change transparently into binary-based deployment through the mechanism of *substitutes*. For any store expression, a *substitute expression* can be registered, which is also just a store derivation expressions. Whenever Nix is asked to normalise a store expression p, it will first try to normalise its substitute, if available. The idea is that the substitute performs the same build as the original expression, but with fewer resources. Typically, this is done by fetching the pre-built contents of the output path of the derivation from the network, or from installation media such as a CD-ROM. This mechanism is generic (policy-free), because it does not force any specific deployment policy onto Nix. Specific policies are discussed in Section 5.

5 Deployment policies

A useful aspect of Nix is that while it is conceptually a source-based deployment system, it can transparently support binary deployment through the substitute mechanism (Section 4.6). Thus, efficient deployment consists of two aspects:

- *Source level*: Nix expressions are deployed to the target system, where they are translated to store expressions and normalised (e.g., through nix-env).
- *Binary level*: Pre-built derivates are made available, and substitute expression are registered on the target system. This latter step is largely transparent to the users. There is no apparent difference between a "source" and a "binary" installation.

Source level deployment is unproblematic, since Nix expressions tend to be small. Typical deployment policies are to obtain sets of Nix expressions packaged into a single file for easier distribution, or to fetch them from a version management system. The latter is useful as it can easily allow automatic upgrades of a system. For instance, we can periodically (from a cron job) update the Nix expressions and build the derivations described by them. Note that due to the successor mechanism this is very efficient for expressions that have not actually changed.

Binary level deployment presents more interesting challenges, since even small Nix expressions can, depending on the variability present in the expressions, yield an

```
subversion = {apacheModule , stdenv}:
  (import ./subversion.nix)
  { clientOnly = false , sslSupport = true
   , apacheModule = apacheModule
   , stdenv = stdenv , ... };

subversion' = {stdenv}:
  [(subversion {apacheModule = true})
   (subversion {apacheModule = false})];

subversion" =
  [(subversion' {stdenv = stdenv-i386-Linux})
   (subversion' {stdenv = stdenv-i386-FreeBSD})];
```

Figure 9: Variant selection

exponentially large set of possible store objects. Also, these store objects are large and may take a long time to build. Thus, we have to decide *which* variants are pre-built, *who* builds them, and *where* they are stored.

Let us first look at the most simple deployment policy: a fixed selection of variants are pre-built, *pushed* onto a HTTP server, from where they can then be *pulled* by clients. To push a derivation, all elements in the resulting closure are packaged (e.g., by placing them into a .tar.gz archive). All of this is entirely automatic: to push the derivations of some expression foo.nix the distributor merely has to issue the command nix-push foo.nix.

The client issues the command nix-pull to obtain a list of available pre-built components available from a pre-configured URL (i.e., the HTTP server). For each derivation available on the server, substitute expressions are registered that (when normalised) will fetch, decompress, and unpack the packaged output path from the server. Note that nix-pull is *lazy*: it will not fetch the packages themselves, just some information about them.

The issue of *which* variants to pre-build requires the distributor to determine the set of variants that are most likely to be useful. For instance, for the Subversion component, it may never be useful to *not* have SSL support, but it may certainly be useful to leave out Apache server support, since that feature introduces a dependency on Apache, which might be undesirable (e.g., due to space concerns). Also, the platform for which to build must be selected. Figure 9 shows how 4 variants of Subversion can be built. The function subversion supplies all arguments of the expression in Figure 2, except apacheModule and stdenv (which determines the build tools, and thus the target platform). The function subversion' uses this to produce two variants, given a stdenv: one with Apache server support, and one without. This function is in turn used by the variable subversion", which calls it twice, with a stdenv for Linux and FreeBSD respectively. Hence, this evaluates to $2 \times 2 = 4$ variants.

Pre-building and pushing to a shared network site merely optimises deployment of common variant selections; it does not preclude the use of variants that are not pre-built. If a user selects a variant for which no substitute exists, the variant will be built locally. Also, input components such as compilers that are exclusively build time dependencies (that is, they appear in the derivation value but not in the closure value) will only be fetched or built when the variant must be built locally.

The tools nix-pull and nix-push are not part of the Nix system as such; they are just applications of the underlying technology. Indeed, they are just short Perl scripts, and can be easily adapted to support different deployment policies. For instance, they can be trivially extended to accommodate multiple servers, which happens when one wants to use pre-built derivates from different sources. A very different scheme is lazy construction, where clients push derivates onto a server if they are not already present there. This is useful if it is not known in advance which derivates will be needed. An example is mass installation of components in a heterogeneous network. In a peer-to-peer architecture each client makes its derivates available to all other clients (that is, it pushes onto itself, and pulls from all other clients). In this case there is no server, and thus, no need to provide central storage that scales in the number of clients.

6 User environment policies

From the discussion of the store in Section 4 it may not be obvious how users are supposed to interact with application components built and managed by Nix. Indeed, we can hardly expect users to type /nix/store/eeeeaf42e56b-subversion-0.32.1/bin/svn when they want to start a program! Clearly, we should hide these implementation details from users.

We solve this problem by *synthesising user environments*. A user environment is the set of applications or programs available to the user through normal interaction mechanisms. In a Unix setting, this means that they appear in a directory in the user's PATH environment variable. In other settings, it might be the presence of icons on the desktop or entries in a Start Menu. Here we show an example of the former. The user has in her PATH variable the path /nix/links/current/bin. /nix/links/current is a symbolic link (symlink) that points to the current user environment *generation*. Generations are symlinks to the actual user environment. They are needed to implement atomic upgrades and rollbacks: when a derivation is added or removed through nix-env, we build the new environment, and then create a generation symlink to it with a number one higher than the previous generation. User

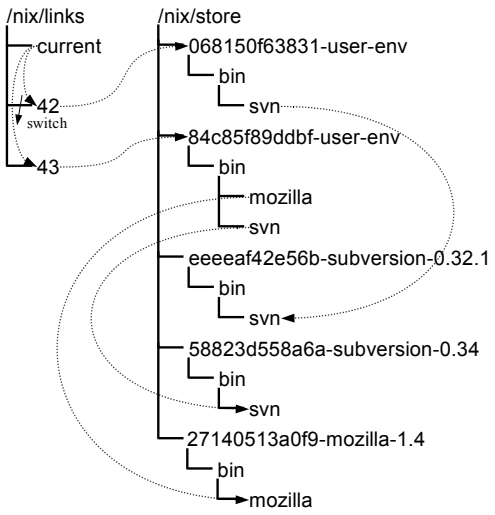

Figure 10: User environments

environments are just sets of symlinks to programs of activated components (similar to, e.g., GNU Stow [4]), and are themselves computed using derivations.

Figure 10 shows what happens when we upgrade Subversion, and add Mozilla in a single atomic action. Dotted lines indicate symlinks. A new environment is constructed in the store based on the current generation (42), the new generation (43) is made to point to it, and finally the current link is switched to point at generation 43. The semantics of the POSIX rename() system call ensures that this is an atomic operation. That is, users and programs always see the old set of activated programs, or the new set, but never neither, both, or a mix. Since old generations are retained, we can atomically downgrade to them in the same manner.

The generation links are the only external links into the store. This means that the only reachable store paths are those in the closure of the targets of the generation links. The closure can be found using the closure values computed in Section 4.5. Since all store paths not in this closure are unreachable, they can be *deleted* at will. This allows Nix to do automatic *garbage collection* of installed components. Nix has no explicit operation to delete a store path—that would be unsafe, since it breaks the integrity of closures containing that path. Rather, it provides operations to remove derivations from the user environment, and to garbage collect unreachable store paths. Store paths reachable only from old generations can be garbage collected by removing the generation links.

This scheme, where a user environment is created for the entire system, is just the simplest user environment policy. The creation of a user environment is just itself a normal derivation. The command nix-env used in Section 3 is just a simple wrapper that automatically creates a derivation, normalises it, and switches the current

```
derivation {
  name = "site-env";
  builder = ./create-symlinks.pl;
  inputs = [
    ((import ./subversion.nix) { ... })
    ((import ./mozilla.nix) { ... })
    ...
  ];
}
```

Figure 11: A Nix expression to build a site-wide user environment

generation to the resulting output path. The build script used by nix-env for environment creation is a fairly trivial Perl script that just creates symlinks to the files in its input closures. A trivial modification is to allow *profiles*—environments for specific users or situations. This can be done by specifying a different link directory (e.g., /home/joe/.nixlinks). Multiple activated versions of the same component in an environment can be accommodated through renaming (e.g., a symlink svn-0.34), which is a policy decision that can be implemented by modifying the user environment build script.

A more interesting extension is *stacked* user environments, where one environment links to the programs in another environment. This is easily accommodated: just as the inputs to the construction of an environment can be concrete components (such as Subversion), they can be other environments. The result is just another indirection in the chain of symlinks. A typical scenario is a 2-level scheme consisting of a site-wide environment specified by the site system administrators, with user-specific environments that augment or override the site-wide environment. Concretely, the site administrator makes a Nix expression as in Figure 11 (slightly simplified) and makes it available on the local network. Locally, a user can then link this site-wide environment into her own environment by issuing the commands

```
nix-env -I site-wide.nix
nix-env -u site-env
```

where site-wide.nix refers to the Nix expression. This will replace any previously installed derivation with the symbolic name site-env. To ensure that changes to the site-wide environment are automatically propagated, these command can be run periodically (e.g., from a cron job), or initiated centrally (by having the administrator remotely execute them on every machine and/or for every user).

Should components in the local environment override those in the site-wide environment? Again, this is a policy decision, and either possibility is just a matter of adapting the builder for the local user environment, for instance to give precedence to derivations called site-env.

7 Related work

Centralised and local package management Package management should be centralised but each machine must be adaptable to specific needs [23]. Local package management is often ignored in favour of centralised package management [16, 14]. In our approach, central configurations can easily be shared and local additions can be made. Any user can be allowed to deviate from a central configuration. Software installation by arbitrary users is discussed in [17]. In [22] policies are introduced that define which installation tasks are permitted. This might be a challenging extension to Nix.

Non-interference Overlapping software packages exist and should not interfere. Typical overlap is caused by multiple versions of an application. It is important that multiple versions can coexist [17], but it is difficult to achieve with current technology [8]. A common approach is to install software packages in separate directories, sometimes called *collections* [23]. In [15], a directory naming scheme is used that restricts the number of concurrent versions and variants of a package. Sharing is in most deployment systems either unsafe due to implicit references, or not supported at all because every application is made completely self-contained [14, 16]. Sharing of data across platforms using a directory structure that separates platform specific from platform independent data is discussed in [14]. Their concern is about diversity in platform, not about diversity in feature sets. As a consequence, in contrast to what they claim, exchange and sharing of packages is not truly safe, as is the case for Nix.

Safe upgrading Many systems ignore this issue [23]. Automatic rollback on failures is discussed in [16]. This turned out to be undesirable in practice because it increased installation time and did not increase consistency. RPM [12] has a notion of transactions: if the installation of a set of packages failed, the entire installation is undone. This is not atomic, which brings the packages being upgraded in an inconsistent state. The approach discussed in [15] uses shortcuts to default package versions, e.g., `emacs` pointing to `emacs-20.2`. This is unsafe because programs may now use `emacs` which initially corresponds to `emacs-20.2`, but after an upgrade points to, e.g., `emacs-20.3`. Separation of production and development software via directories is discussed in [14]. Once an application has been fully tested under the development tree it is turned into production. This requires recompilation because path names will change and may cause errors. Consequently, the approach is not really safe.

Garbage collection In [17] an approach for removing old software is discussed. This approach is not safe because there exists no static model of software dependencies. Basically, software is removed and then it is discovered whether other software fails. If so, the deletion is rolled-back. Moreover, their approach is not portable. The Nix approach is both safe and portable.

Dependency analysis In [17] a pointer scanning mechanism is discussed similar to ours. Their approach is not safe, making false positives very likely to occur. In [20] a dependency analysis tool for dynamic libraries is discussed. In Nix this information is already available when an application is installed. Furthermore, Nix is not restricted to detect dependencies on shared libraries only.

Safety In [22] common *wrong* assumptions of package managers are explained, including: i) package installation steps always operate correctly; ii) all software system configuration updates are the result of package installation. In Nix, software gets installed safely, without affecting the environment. Thus, in contrast to many other systems, Nix will never bring a system in an unstable state. Unless a system administrator really wants to mess things up, all upgrades to the Nix store are the result of package installation. Safe testing of applications outside production environments is discussed in [17, 14]. In [13] it is confirmed that software should be installed in private locations because software packages often have overlapping content. Overlap between packages turns out to be a very common cause of installation problems. In Nix, such packages can safely coexist.

Packaging In [19] a generic packaging tool for building bundles (i.e., collections of products that may be installed as a unit) is discussed. Source tree composition [10] is an alternative technique for automatic producing bundles from source code components. However, these bundling approaches do not cater for sharing of components *across* bundles.

8 Conclusions and future work

Seemingly simple tasks such as installing or upgrading an application often turn out to be much harder than they should be. Unexpected failures and the inability to perform certain actions affect users of all levels of computing expertise. In this paper we have pinpointed a number of causes of the deployment malady, and described the Nix system that addresses these by using cryptographic hashes to enforce uniqueness and isolation between components.

Nix is free software and is available online [5]. It is being used with good results to deploy various components to several different Linux-based operating systems. A common problem with these systems is that they often differ in subtle ways that cause packages built on one system to fail on another, e.g., because of C library incompatibilities. Our Nix components, however, are completely boot-strapped, that is, they are built using only build tools, libraries, etc., that have themselves been built using Nix,

and do not rely on components outside of the Nix store (other than the running kernel). Using our reliable dependency analysis, any required libraries and other components are deployed also. Thus, they just "work".

There are a number of interesting issues remaining. Of particular interest is our expectation that Nix will permit *sharing* of derivations between users. That is, if user A has built some derivation, and user B attempts to build the same derivation, B can transparently reuse A's result. Clearly, using code built by others is not safe in general, since A may have tampered with the result. However, our use of cryptographic hashes can make this safe, since the hash includes all build inputs, and therefore completely characterises the result.

The problems of dependency identification and dealing with variants also plague build managers such as Make [11]. We believe that with suitable developer facilities Nix can be used to replace these more low-level software configuration management tools as well.

The fundamental limitation to Nix's dependency checking is that it won't prevent undeclared references to artifacts outside of the store. For instance, if a builder calls /bin/sh, we won't know about it. Ideally, *all* components would be stored in the Nix store; there would be no "global" locations such as /bin. We have been using User Mode Linux [7] to experiment with a system with this property, but more experience is needed.

Another limitation is that Nix does not handle component *state*. For instance, a component upgrade may require configuration files to be updated. Even if the component does this automatically, it precludes a rollback. Finding an elegant way to deal with this will be a challenge.

References

[1] Autoconf. http://www.gnu.org/software/autoconf/.

[2] FreeBSD Ports Collection. http://www.freebsd.org/ports/.

[3] Gentoo Linux. http://www.gentoo.org/.

[4] GNU Stow. http://www.gnu.org/software/stow/.

[5] Nix. http://www.cs.uu.nl/groups/ST/Trace/Nix.

[6] Subversion. http://subversion.tigris.org/.

[7] User Mode Linux. http://user-mode-linux.sourceforge.net/.

[8] E. Anderson and D. Patterson. A retrospective on twelve years of LISA proceedings. *Proceedings of the 13th Systems Administration Conference (LISA 1999)*, pages 95–107, Nov. 1999.

[9] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, number 28/6 in SIGPLAN Notices, pages 197–206, June 1993.

[10] M. de Jonge. Source tree composition. In *Seventh International Conference on Software Reuse*, number 2319 in Lecture Notes in Computer Science. Springer-Verlag, Apr. 2002.

[11] S. I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–65, 1979.

[12] E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley and Sons, 2003.

[13] J. Hart and J. D'Amelia. An analysis of RPM validation drift. In *Proceedings of the 16th Systems Administration Conference (LISA-2002)*, pages 155–166. USENIX Association, Nov. 2002.

[14] K. Manheimer, B. A. Warsaw, S. N. Clark, and W. Rowe. The Depot: A framework for sharing software installation across organizational and UNIX platform boundaries. *Proceedings of the 4th Systems Administration Conference (LISA 1990)*, pages 37–46, Oct. 1990.

[15] T. Oetiker. SEPP: Software installation and sharing system. *Proceedings of the 12th Systems Administration Conference (LISA 1998)*, pages 253–259, Dec. 1998.

[16] K. Oppenheim and P. McCormick. Deployme: Tellme's package management and deployment system. *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, pages 187–196, Dec. 2000.

[17] J. P. Rouillard and R. B. Martin. Depot-lite: a mechanism for managing software. *Proceedings of the 8th Systems Administration Conference (LISA 1994)*, pages 83–91, 1994.

[18] B. Schneier. *Applied Cryptography*. John Wiley and Sons, second edition, 1996.

[19] C. Staelin. mkpkg: A software packaging tool. *Proceedings of the 12th Systems Administration Conference (LISA 1998)*, pages 243–252, Dec. 1998.

[20] Y. Sun and A. L. Couch. Global impact analysis of dynamic library dependencies. *Proceedings of the 15th Systems Administration Conference (LISA-2001)*, pages 145–150, Nov. 2001.

[21] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceeedings of WICSA 2001*, August 2001.

[22] V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa, and Z. Liang. An approach for secure software installation. In *Proceedings of the 16th Systems Administration Conference (LISA-2002)*, pages 219–226. USENIX Association, Nov. 2002.

[23] W. C. Wong. Local disk depot: customizing the software environment. *Proceedings of the 7th Systems Administration Conference (LISA 1993)*, pages 49–53, Nov. 1993.