Nix: A Safe and Policy-Free System for Software Deployment

Eelco Dolstra
Merijn de Jonge
Eelco Visser

Address:
Eelco Dolstra
eelco@cs.uu.nl
http://www.cs.uu.nl/~eelco

Merijn de Jonge
mdejonge@cs.uu.nl
http://www.cs.uu.nl/~mdejonge

Eelco Visser
visser@acm.org
http://www.cs.uu.nl/~visser

Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht

Nix: A Safe and Policy-Free System for Software Deployment

Eelco Dolstra, Merijn de Jonge and Eelco Visser
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
{eelco, mdejonge, visser}@cs.uu.nl

December 9, 2004

Abstract

Existing systems for software deployment are neither safe nor sufficiently flexible. Primary safety issues are the inability to enforce reliable specification of component dependencies, and the 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 set of installed software components 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 [15], e.g., by overwriting the files of those components. In particular, it should be possible to have multiple versions and variants of a component installed at the same time. 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 interference between components and that end-user customisation has only been slightly examined [6]. 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 primary features of Nix are:

- Concurrent installation of multiple versions and variants

- Atomic upgrades and downgrades

- Multiple user environments

- Safe dependencies

- Complete deployment

- Transparent binary deployment as an optimisation of source deployment

- Safe garbage collection

- Multi-level package management (i.e. different levels of centralised and local package management)

- 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. Our experience with Nix is discussed in Section 7. Section 8 compares Nix to previous work. We conclude with some remarks on future work in Section 9.

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 (http://subversion.tigris.org/). 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 [11] can use the ldd tool at packaging time 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 [24]. The Subversion component has several optional features, such as whether we want support for OpenSSL encryption and authentication, whether only a Subversion client should be built, and whether an Apache server module should be built so that Subversion can act as a WebDAV server. 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 contain 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 nec-

4

essary to upgrade shared components such as shared libraries first. If they are not backwards compatible, then there will be a timing window in which components that use them fail to work properly.

Identification 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 manually is unfortunate, since binary deployment can be considered an optimisation of source deployment because it uses fewer resources on the target system. Ideally, the creation of binary packages 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 they are shared libraries, then their full paths (e.g., /usr/lib/libssl.so.0.9.6) will be stored in the resulting Subversion executables, causing these build time dependencies to 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 machine-local policies.

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[1] system are the *Nix store* for storing isolated installations of components; *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 a generic means of *sharing* build results between machines. These ingredients provide *mechanisms* for implementing a wide variety of deployment *policies*. In this section we give a high-level 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 interacts with the applications installed on a computer through a certain interface. This may be the *start menu* on Windows and other desktop environments, or the PATH environment variable in command-line interfaces on Unix-like systems. These interfaces form what we call the *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 the *installation space*.

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

---

[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.

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

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://.../subversion-0.32.1.tgz;
    md5 = "b06717a8ef50db4b...";
  };

  # Pass these to the builder.
  inherit clientOnly apacheModule sslSupport;
          stdenv openssl httpd db4;
}
```

Figure 1: Subversion component (subversion.nix)

```
stdenv = import ...;
openssl = import ...;
... # other component definitions

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

Figure 2: Subversion composition (pkgs.nix)

Derivation The body of the expression is formed by calling the primitive function derivation with an *attribute set* {key=value;...}. The set contains two attributes required by the derivation function: the builder attribute indicates a script that builds the component, while the system attribute specifies the target platform on which the build is to be performed. The other attributes define values for use in the build process (such as dependencies) and are passed to the build script as environment variables. The name attribute is a symbolic identifier for use in the high-level user interface; it does not necessarily uniquely identify 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. The syntax for functions is {k_1, ..., k_n}: body, which defines a function that expects to be called with an attribute set containing attributes with names k_1 to k_n. Thus, the Subversion expression is parameterised with expressions describing the components on which it depends (e.g., openssl, httpd, stdenv), options that select features (e.g., clientOnly, sslSupport), and a utility (fetchurl). The stdenv component provides 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 2 instantiates the Subversion expression by assigning values to its parameters.

A subtle but important difference with most component formalisms is that in Nix we explicitly describe not just components but also compositions of components. For instance, an RPM spec file specifies how to build a component, but not its dependencies. It merely states fairly weak conditions on the expected build environment ("a package called glibc-2.3.2 should be present"). Thus, a spec file is always

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.

3.2 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 1 shows a Nix function that returns variants of the Subversion system, based on certain parameters; it features most typical constructs of the language. Figure 2 shows a call to this function. We will use these examples to explain the elements of the language.

```
buildInputs="$openssl $db4 $httpd"
# Bring in GCC etc., 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

...

tar xvfz $src
cd subversion-*
./configure --prefix=$out $extraFlags
make
make install
```

Figure 3: Subversion build script (builder.sh)

incomplete, so there is no way to uniquely specify concrete components. The Subversion Nix expression in Figure 1 is similarly incomplete, but the composition in Figure 2 provides the whole picture — information on how to build not just Subversion, but also all of its dependencies.

The value of the src attribute is another example of functional computation. Its value is the result of a call to the function fetchurl (passed in as an argument of the Subversion function) that downloads the source from a specific URL and verifies that it has the right MD5 checksum.

Assertions In order to restrict the values that can be passed as parameters, a function can state assertions over the parameters. For example, the db4 database is needed only when a local server is implemented. Also, *consistency* between components can be enforced. For instance, if both SSL and Apache support are enabled, then 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 enforced, link errors could result.

Build Script When a derivation is built, the build script indicated by the builder attribute is invoked. As stated above, attributes of the derivation are passed through environment variables to the builder. In the case of attributes that refer to other derivations (i.e. dependencies), the corresponding environment variables contain the paths at which they are stored. Nix ensures that such dependencies are built prior to the invocation of the builder, so the build script can assume that they are present. The special variable out conveys to the builder where it should store the build result. Figure 3 shows the build script for Subversion. The largest part of the script is used to compute the configuration flags based on the features selected for the Subversion instance. 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.

3.3 User Environments

A Nix *user environment* consists of a selection of applications from the store currently relevant to a user. "Users" can be human users, but also system users such as daemons and servers that 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 do not 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 (e.g., [4]) 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. 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 different 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.

For instance, to add the Subversion component in Figure 2 to the current user environment, we do:

```
$ nix-env -f pkgs.nix -i subversion
```

where pkgs.nix is the file containing the definition in Figure 2. This will build Subversion and create a new user environment, based on the old one, to which Subversion has been added. If an expression for a new Subversion release comes along, we can upgrade as follows:

```
$ nix-env -f pkgs.nix -u subversion
```

which likewise creates a new user environment, based on the old one, in which the old Subversion component has been replaced by the new one. However, the old user environment and the components included in it are retained, so it is possible to return to the old situation if necessary:

```
$ nix-env --rollback
```

There is no operation to physically remove components from the system. They can only be removed from a user environment, e.g.,

```
$ nix-env -e subversion
```

creates a new user environment from which the links to Subversion have been removed. However, storage space can be reclaimed by periodically running a garbage collector:

```
$ nix-collect-garbage
```

which removes any component not reachable from any user environment. (Therefore it is necessary to periodically prune old user environments, e.g., once we find that we do not need to roll back to old ones). Garbage collection is safe because we know the full dependency graph between components (Section 4).

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 copied to a Nix store on another machine. For this purpose, Nix provides support for transparently maintaining a collection of pre-built components on some shared medium such as an FTP site or an installation CD-ROM. After building a component in the store it can be *pushed* to the shared medium.

For instance, the installation and upgrade operations above perform an installation from source. This is generally not desirable since it is slow. However, it is possible to safely and transparently re-use pre-built components from a shared resource such as a network repository. For instance, a component distributor or system administrator can pre-build components, then *push* (upload) them to a server using PUT requests:

```
$ nix-push http://example.org/cache \
    pkgs.nix subversion
```

This will build Subversion (if necessary) and upload it and all its dependencies to the indicated site. A user can then make Nix aware of these:

```
$ nix-pull http://example.org/cache
```

Subsequent invocations of nix-env -i / -u will automatically use these *if* they are exactly equal to what the user is requesting to be installed. That is, if the user changes any sources, flags, and so on, the pre-built components will *not* be used, and Nix will revert to building the components itself. Thus, Nix is both a source and binary-based deployment system; 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.

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 store

The two main design goals of the Nix system are to *support concurrent variants*, and to *ensure complete dependency information* which is necessary to support *completeness* (the deployment process should transmit all dependencies necessary for the correct operation of a component). It turns out that the solutions to these goals are closely related. 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 involved in building the component.*

---

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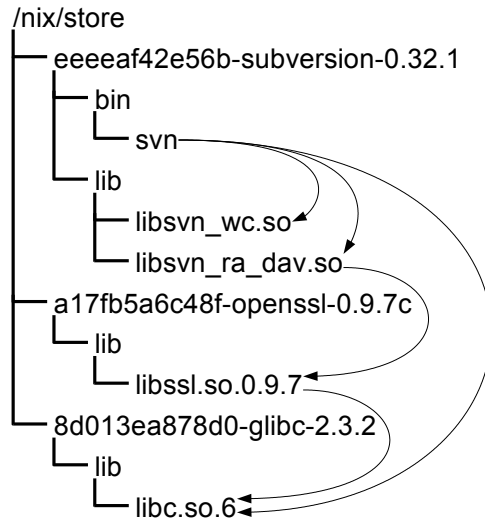

Figure 4: The store

Thus, each object in the store has a unique name, so that variants can co-exist. These names are called *store paths*. In Autoconf [1] terminology, each component has a unique prefix. The file system 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 4 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.

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

9

problem of retained dependencies (discussed in Section 2): it is not necessary to declare explicitly those 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 using them might cause other components to be referenced that are missing on the target system.

4.2   Building components

So how do we build components from Nix expressions? This 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 identified uniquely. Since Nix expressions can import other expressions scattered all 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 could retarget Makefiles to use Nix as a backend).

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. Store expressions describe how to build one or more store paths. *Realisation* of a store expressions means making sure that all those paths are present in the store.

Derivation store expressions describe the building of a single store component. They describe all inputs to the build process: other store expressions that must be realised first (build time dependencies), the build platform, the build script (which is one of the dependencies), and environment variable bindings. These are computed from calls to the derivation function in the Nix expression language by recursively translating all input derivations to derivation store expressions, copying source files to the store, and adding all attributes as environment variable bindings.

To perform the build action described by a derivation, the following steps are taken:

1. Locks are acquired on the output path (the store path of the component being built) to ensure correctness in case of parallel invocations of Nix.

2. Input store expressions are realised. This ensures that all file system inputs are present.

3. The environment is cleared and initialised to the bindings specified in the derivation.

4. The builder is executed.

5. If the builder was executed successfully, we build a *closure store expression* that describes the resulting closure, i.e., the output path and all store paths directly or indirectly referenced by it. We do this by scanning every file in the output path for occurrences of the cryptographic hashes in the input store paths. For instance, when we build Subversion, the path /nix/store/a17fb5a...-openssl-0.9.7c is passed as an input. After the build, we find that the string a17fb5a... occurs in the file libsvn_ra_dav.so (as shown in Figure 4). Thus, we find that Subversion has a retained dependency on OpenSSL. Build time dependencies carried over to runtime are detected automatically in this way. (This approach is discussed in more detail in [9]).

6. The closure expression is written to the store.

The command nix-instantiate translates a Nix expression to a store expression:

```
$ nix-instantiate pkgs.nix
/nix/store/ce87...1095-subversion.store
```

The command nix-store --realise realises a derivation store expression, returning the resulting closure store expression:

```
$ nix-store --realise \
   /nix/store/ce87...1095-subversion.store
/nix/store/ab1f...77ef.store
```

Nix users do not generally have to deal with store expressions. For instance, the nix-env command hides them entirely — the user interacts only with high-level Nix expressions, which is really just a fancy wrapper around the two commands above. However, store expressions are important when implementing deployment policies. Their relevance is that they give us a way to uniquely identify a component both in source and binary form, through the derivation and closure store expression, respectively. This can be used to implement a variety of deployment policies, as we will see in Section 5.

A crucial operation for deployment is to query the set of store paths referenced by a store expression. This is the set of paths that must be copied to another system to ensure that it can realised there. For instance, for the derivation above we get:

```
$ nix-store --qR \
   /nix/store/ce87...1095-subversion.store
/nix/store/ce87...1095-subversion.store
/nix/store/d1bc...0aa1-builder.sh
/nix/store/f184...3ed7-gcc.store
/nix/store/f199...0719-bash.store
...
```

That is, this set includes the derivation store expressions for building Subversion itself and its direct and indirect dependencies, a closure store expression for the builder, and so on.

On the other hand, for the closure we get:

```
$ nix-store --qR \
   /nix/store/ab1f...77ef.store
/nix/store/ab1f...77ef.store
/nix/store/eeee...e56b-subversion-0.32.1
/nix/store/a17f...c48f-openssl-0.9.7c
/nix/store/8d01...78d0-glibc-2.3.2
...
```

This set only includes the closure store expression itself and the component store paths it references.

4.3 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, unnecessary 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-source 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 path, a *substitute expression* can be registered, which is also just a store derivation expression. Then, whenever Nix is asked to realise a closure that contains path p, and p does not yet exist, it will first try to build 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 4.3). 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 built (e.g., through nix-env).

- *Binary level*: Pre-built derivates are made available, and substitute expressions 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 (e.g., from a cron job) update the Nix expressions and build the derivations described by them. Note that any subexpressions that have not changed do not need to be rebuilt.

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

Figure 5: Variant selection

Binary level deployment presents more interesting challenges, since even small Nix expressions can, depending on the variability present in the expressions, yield an 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 built) 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 5 shows how 4 variants of Subversion can be built. The function subversion supplies all arguments of the expression in Figure 1, 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 from source. 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 applications of the underlying technology. Indeed, they are just short Perl scripts, and can easily be adapted to support different deployment policies. For instance, an entirely different policy 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 scaling in the number of clients.

6 User environment policies

The use of cryptographic hashes in store paths gives us reliable identification of dependencies and non-interference between components, but we can hardly expect users to type, e.g., /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, which in a Unix setting means that they appear in a directory in the user's PATH environment variable. 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 environ-
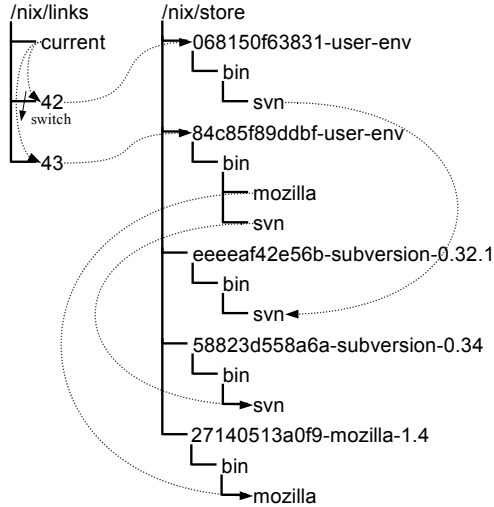
Figure 6: User environments

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

Figure 7: A Nix expression to build a site-wide user environment (site-wide.nix)

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

This is illustrated in Figure 6 (dotted lines denote symlinks), where the current symlink points to generation 42, which is in turn a symlink to a user environment in the store. The user environment is simply a tree of symlinks to activated components. Hence, the path /nix/links/current/bin/svn indirectly refers to /nix/eeee...-subversion-0.31.1/bin/svn.

Figure 6 also shows what happens when we upgrade Subversion, and add Mozilla in a single atomic action. 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. 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 itself a normal derivation, and the command nix-env used in Section 3 is a simple wrapper that automatically creates a derivation, builds it, and switches the current generation to the resulting output path. The build script used by nix-env for environment creation is a fairly trivial Perl script that creates symlinks to the files in its input closures. A simple 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). Also, multiple versions of the same program 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 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 7 (slightly simplified) and makes it avail-

able on the local network. Locally, a user can then link this site-wide environment into her own environment by doing

```
nix-env -f site-wide.nix -i 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 commands 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.

Server configurations User environments (contrary to what the term implies) can not only be used to specify environments for specific users, but also for specific tasks or processes. In particular, they can be used to specify complete *server configurations*, which includes not only the software components constituting some server, but also its configuration and other auxiliary files. Consider, for instance, an Apache/Subversion server (the Subversion server runs as a module on top of Apache). It consists of several components that are rather picky about specific dependencies, e.g., Apache, Subversion, ViewCVS, Python, and Perl, but also our repository management CGI scripts, static HTML documents and images, the Apache httpd.conf configuration file, SSL private keys, and so on. Since these are also components (just not necessarily executable components) they can be managed using Nix.

Figure 8 shows a (simplified) Nix expression for an Apache/Subversion server. It takes a single argument that specifies whether a test or production server is to be built. The builder produces a component consisting of an Apache configuration file, and a control script to start and stop the server. The builder generates these by substituting values such as the desired port number and the paths to the Apache and Subversion components into the given source files.

Now, given a simple script upgrade-server (not shown here) that uses nix-env -u to build the new server configuration, stop the server running in the old generation, and start the new one, we can easily instantiate new server configurations by editing source files such as httpd.conf.in, and calling upgrade-server. For instance, the command upgrade-server test instantiates the Nix expression by calling it with a

```
{productionServer}:
derivation {
  builder = ./builder.sh;
  configuration = ./httpd.conf.in;
  controller = ./ctl.sh.in;
  portNumber = if productionServer
              then 80 else 8080;
  inherit (import ...) httpd subversion;
}
```

Figure 8: A Nix expression to build a Subversion server

false argument, thus producing a test server. If this is found to work properly, we can issue upgrade-server production to upgrade the production server. nix-env --rollback can be used to go back to the previous generation, if necessary.

The server is started using the script controller.sh which is part of the server configuration component. It initialises PATH to point to a specific set of components. This means that the server configuration is self-contained: it does not depend on anything not explicitly specified in the Nix expression. Such a configuration is therefore pretty much immune to external configuration changes, and can be relatively easily transferred to another machine.

The only thing not under Nix control here is *state* — things that are modified by the server, e.g., the actual Subversion repositories and user account databases.

Thus, Nix can be used for the deployment of not just software components, but also complete system configurations — the domain of tools such as Cfengine [7]. Note that Cfengine declaratively specifies *destructive changes* to be performed to realise a desired configuration. This makes it hard to easily run several configurations in parallel on the same machine, or to switch back and forth between configurations. Also, Cfengine is typically not used to manage the software components on a machine (although this is possible, e.g., by installing the appropriate packages in an Cfengine action [20]).

7 Experience

We have applied Nix to a number of problem domains.

Software deployment We have "nixified" 180 or so existing Unix packages, including large ones such as Mozilla Firefox with all its dependencies (which

includes the C compiler, basic Unix tools, X11, etc.). They are prebuilt for Linux and made available through the push/pull mechanism described in Section 5.

The fundamental limitation to Nix's dependency checking is that it will not prevent undeclared dependencies on components outside of the store. For instance, if a builder calls /bin/sh, we have no way to detect this. To minimise the probability of such undeclared dependencies, we use patched versions of gcc, ld, and glibc that refuse to use header files and libraries outside of the Nix store. In our experience this works quite well. For instance, the prebuilt Nix packages work on a variety of Linux distributions— evidence that no (major) external components are used. A common problem with these distributions 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. However, our Nix components 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".

The ability to very rapidly perform rollbacks is often a life-saver. For instance, it happens quite frequently that we attempt to upgrade some bleeding-edge software package, only to discover that it doesn't work quite as well as the previous version (or not at all!). A simple nix-env --rollback saves the day. In most package managers, recovery would be much harder, since we would have to know exactly what the previous configuration was, and we would have to have a way to re-obtain the old versions of the packages that were just upgraded.

Service deployment As described in Section 6, Nix can be used for the deployment of not just software components, but also complete configurations of system services. For instance, our department's Subversion server is managed in this way. The main advantages are that it is very easy to run multiple instances of a service (e.g., for testing — and the test server will in no way interfere with the production server!), that it is easy to move a service to another machine since we have full dependency information, and again that we can rollback to earlier versions.

Build farms It is a good software engineering practice to build software systems continuously during the development process [13]. In addition, if software is to be portable, it should be built on a variety of machines and configurations. This requires a *build farm* — a set of machines that sit in a loop building the latest version obtained from the version management system. Build farms are also important for *release management* — the production of software releases — which must be an automatic process to ensure reproducibility of releases, which is in turn important for software maintenance and support.

The management of a build farm is often highly time-consuming. For instance, if the component being built in the build farm requires (say) Automake 1.7, we must install that version of Automake on each machine in the build farm. If at some point we need a newer version of Automake, we again must go to each machine to perform the upgrade. So maintaining a build farm scales badly. Worse, there may be conflicting dependencies (e.g., some other component in the build farm may only work with Automake 1.6).

Such management of dependencies is exactly what Nix is good at, so we have implemented a build farm on top of Nix. The main advantages over other build farms (e.g., [12]) are:

- The Nix expression language makes it easy to describe the build tasks, along with their dependencies.

- Nix ensures that the dependencies are installed on each machine in the build farm.

- The hashing scheme ensures that identical builds (e.g., of dependencies) are performed only once.

- In Nix, each derivation has a system attribute that specifies on what kind of platform the derivation is to be performed (e.g., i686-linux). If the attribute does not match the type of the platform on which Nix is run, Nix can automatically distribute the derivation to a different machine of the intended platform type, if one exists. All inputs to the derivation are copied to the store of the remote machine, Nix is run on the remote machine, and the result is copied back to the local store. Thus, dealing with multi-platform builds is fairly transparent: we can write a Nix expression specifying derivations on a variety of platforms and run it on a arbitrary machine. There is no need to schedule the build separately on each machine.

- The resulting builds can be used immediately by other developers since they are made available through nix-push.

A downside to a Nix-based build farm is that installing a package through Nix differs from the "native" way of installing a package on existing platforms (e.g., by installing an RPM on a Red Hat machine). Thus it is difficult for a Nix build farm to verify whether a package works when built from source in the native way. However, on Linux systems, we can in fact build native packages (such as RPMs) without affecting the host system by using User-Mode Linux [5] in Nix derivations. In fact, this fits in quite well. For instance, the synthesis of the UML disk images for the various platforms for which we build packages is just a normal Nix derivation that creates an Ext2 file system from an arbitrary set of RPMs constituting a Linux distribution.

8   Related work

Centralised and local package management Package management should be centralised but each machine must be adaptable to specific needs [26]. Local package management is often ignored in favour of centralised package management [19, 17]. 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 [21]. In [25] policies are introduced that define which installation tasks are permitted. This might be a challenging extension to Nix. Modules [14] makes software deployment more transparent by abstracting from the details of software deployment. Application-specific deployment details are captured in "module-files", which can be shared between large-scale distributed networks, similar to Nix expressions. Modules lack the safety properties of Nix. As a result, correct operation of typical deployment tasks, as discussed in Section 2, cannot be guaranteed.

Non-interference Software packages should not interfere with each other. Typical interference is caused by attempting to have multiple versions of a component installed. It is important that multiple versions can coexist [21], but this is difficult to achieve with current technology [6]. A common approach is to install software packages in separate directories, sometimes called *collections* [26]. In [18], 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 [17, 19]. Sharing of data across platforms

using a directory structure that separates platform specific from platform independent data is discussed in [17], which is concerned with diversity in platform, not diversity in feature sets. As a consequence however, exchange and sharing of packages is not truly safe, as is the case for Nix.

Safe upgrading Many systems ignore this issue [26]. Automatic rollback on failures is discussed in [19]. This turned out to be undesirable in practice because it increased installation time and did not increase consistency. RPM [11] has a notion of transactions: if the installation of a set of packages failed, the entire installation is undone. This is not atomic, so the packages being upgraded are in an inconsistent state during the upgrade. The approach discussed in [18] 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 [17]. 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 [21] an approach for removing old software is discussed. Basically, after software is "removed" by making the directory unreadable, one verifies whether other software fails by running it. If so, the deletion is rolled back by making the directory readable again. This is unsafe because the test executions may not reveal every dependency, and because a time window is introduced during which some components do not work.

Dependency analysis However, the same paper also describes a pointer scanning mechanism similar to ours: component directories are scanned for the names of other component directories (e.g., tk-3.3). However, such names are not very unique (contrary to cryptographic hashes) and may lead to many false positives. Also, component dependencies are scanned for *after* the component has been made unreadable, not before. In [23] a dependency analysis tool for dynamic libraries is discussed. In Nix this information is already available when an application is installed, and Nix is not restricted to detecting dependencies on shared libraries only. Vesta [16] is a system for configuration management that supports automatic dependency detection. Like Nix, it detects only dependencies that are actually needed, and dependen-

cies are complete, i.e., every aspect of the computing environment is described and controlled by Vesta.

Safety In [25] 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 [21, 17]. In [15] it is confirmed that software should be installed in private locations to prevent interference between software packages. Interference turns out to be a very common cause of installation problems. In Nix, such packages can safely coexist.

Packaging In [22] a generic packaging tool for building bundles (i.e., collections of products that may be installed as a unit) is discussed. Source tree composition [8] is an alternative technique for automatically producing bundles from source code components. However, these bundling approaches do not cater for sharing of components *across* bundles.

9 Conclusion

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. It is successfully used to deploy software components to several different operating systems, to manage server configurations, and to support a build farm.

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 [10]. We believe that (with some extensions) Nix can be used to replace these more low-level software configuration management tools as well.

Availability Nix is free software and is available online at http://www.cs.uu.nl/groups/ST/Trace/Nix.

10 Author Information

Eelco Dolstra is a PhD student in the Software Technology group at Utrecht University, where he also obtained his Master's degree on the integration of functional and strategic term-rewriting languages. His research focuses on dealing with variability in software systems and configuration management, in particular software deployment.

Merijn de Jonge obtained his PhD degree from the University of Amsterdam in the area of software reuse. After a postdoc position at Eindhoven University, he currently works as a postdoc at Utrecht University. His research interests include software reuse, configuration and build management, software variability, generative programming, component-based software development, program transformation, and language-centered software engineering.

Eelco Visser studied computer science at the University of Amsterdam where he obtained Master's and PhD degrees in the area of syntax definition and language processing. As a postdoc at the Oregon Graduate Institute in Portland he laid the foundation for the Stratego program transformation language. He is currently an assistant professor in the Software Technology group at Utrecht University where he leads research projects into program transformation and software configuration and deployment.

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

[6] E. Anderson and D. Patterson. A retrospective on twelve years of LISA proceedings. *Proceedings of the 13th Systems Administration Conference (LISA '99)*, pages 95–107, November 1999.

[7] Mark Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995.

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

[9] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

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

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

[12] Mozilla Foundation. Tinderbox. http://www.mozilla.org/tinderbox.html.

[13] Martin Fowler. Continuous integration. http://www.martinfowler.com/articles/continuousIntegration.html.

[14] J. L. Furlani and P. W. Osel. Abstract yourself with modules. *Proceedings of the 10th Systems Administration Conference (LISA '96)*, pages 193–204, September 1996.

[15] John Hart and Jeffrey D'Amelia. An analysis of RPM validation drift. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*, pages 155–166. USENIX Association, November 2002.

[16] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta approach to software configuration management. Technical Report Research Report 168, Compaq Systems Research Center, March 2001.

[17] 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 '90)*, pages 37–46, October 1990.

[18] T. Oetiker. SEPP: Software installation and sharing system. *Proceedings of the 12th Systems Administration Conference (LISA '98)*, pages 253–259, December 1998.

[19] 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, December 2000.

[20] D. Ressman and J. Valdés. Use of Cfengine for automated, multi-platform software and patch distribution. *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, pages 207–218, December 2000.

[21] J. P. Rouillard and R. B. Martin. Depot-lite: a mechanism for managing software. *Proceedings of the 8th Systems Administration Conference (LISA '94)*, pages 83–91, 1994.

[22] C. Staelin. mkpkg: A software packaging tool. *Proceedings of the 12th Systems Administration Conference (LISA '98)*, pages 243–252, December 1998.

[23] 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, November 2001.

[24] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceeedings of WICSA 2001*, August 2001.

[25] 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 '02)*, pages 219–226. USENIX Association, November 2002.

[26] W. C. Wong. Local disk depot: customizing the software environment. *Proceedings of the 7th Systems Administration Conference (LISA '93)*, pages 49–53, November 1993.