Imposing a Memory Management Discipline on Software Deployment

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

Merijn de Jonge
Eindhoven University of Technology, P.O. Box 513
5600 MB Eindhoven, The Netherlands
m.de.jonge@tue.nl

Abstract

The deployment of software components frequently fails because dependencies on other components are not declared explicitly or are declared imprecisely. This results in an incomplete reproduction of the environment necessary for proper operation, or in interference between incompatible variants. In this paper we show that these deployment hazards are similar to pointer hazards in memory models of programming languages and can be countered by imposing a memory management discipline on software deployment. Based on this analysis we have developed a generic, platform and language independent, discipline for deployment that allows precise dependency verification; exact identification of component variants; computation of complete closures containing all components on which a component depends; maximal sharing of components between such closures; and concurrent installation of revisions and variants of components. We have implemented the approach in the Nix deployment system, and used it for the deployment of a large number of existing Linux packages. We compare its effectiveness to other deployment systems.

1. Introduction

As any computer user knows, *software installation* is a fragile process that fails surprisingly often for seemingly trivial reasons: the component being installed requires another component that is not installed; the installed component is not the right version or variant; the installation process automatically installs the required component, overwriting a newer version already present with an older version, causing other applications to fail; and so on.

Software deployment is the collection of activities concerned with transferring software components from producer to consumer and maintenance of a consumer installation [5]. This includes installation, but also upgrading and de-installation of software. Many software deployment failures are the result of an unsound treatment of the *dependencies* between the components being deployed. Dependencies on other components are not declared explicitly, causing an incomplete reproduction of the environment necessary for proper operation of the components. Furthermore, dependency information that *is* declared, is often not precise enough, allowing incompatible variants of a component to be used, or causing interference between such variants.

In this paper, we present a simple and effective solution to such deployment problems. In Section 2 we analyse the problems that occur in software deployment. We then show in Section 3 that these deployment hazards correspond to pointer hazards in memory models of programming languages. In modern programming languages these hazards are countered by a memory management discipline, which is absent in deployment. Based on this analysis we have developed an analogous discipline for software deployment. The discipline is generic, that is, not specific for a platform, programming language, or build technology. Only a few sanity requirements are imposed on components.

Sound deployment requires deploying *closures* containing all components needed for the operation of a software system (Section 4). The computation of such closures requires complete and precise dependency information, which is achieved by imposing an appropriate *pointer discipline* on names in the file system (Section 5). Deployment of closures is similar to persistence in memory management, i.e., the migration of data from one address space to another. To prevent the occurrence of address clashes as a result of migration we impose a regime of *cryptographic hashes* which *exactly identify components* (Section 6). Identification is based on the contents of a component rather than just its name and version number. Since these *unique product codes* distinguish variants of components, the discipline supports concurrent installation of revisions and variants of components, reproducible installation, and maximal sharing of common components between closures.

This deployment discipline forms the basis for the Nix deployment system (Section 7) which ensures isolation between components. On top of the basic operations for maintaining the component store, Nix provides reliable caching of deployed components, automatic and safe garbage collection of unused components, and transparent deployment of source and binary components. In Section 8 we discuss our experience with this implementation, which includes the deployment of a large number of existing Linux packages along with a complete build environment. We compare its effectiveness to other package managers.

We discuss related work in Section 9, and end with conclusions and suggestions for future work in Section 10.

2. Deployment hazards

Software deployment is the set of activities involved in maintaining software applications on computer systems of end users. The goal is to correctly reproduce software from the system of the developer or distributor—where the software presumably has been tested and found to work—onto the systems where the software is to be used. If this reproduction is correct, then the software will also work on the target system. Reproduction is incorrect if parts of the software are left out of the deployment, are overridden on the target system by others, or are misconfigured. Then we can no longer make any guarantees about the software: it may behave differently, or it may not work at all.

Software deployment involves assembling and installing software distributions or *packages*: a collection of files and directories containing programs, libraries, documentation, etc. A package abstracts from its internal structure via interfaces: the functionality implemented by a package forms a *provides* interface, and the functionality that it requires constitutes a *requires* interface (the *dependencies*). Since packages have a clear analogy with software components [15], we address software deployment from a component-based software engineering (CBSE) perspective: packages are components; software deployment is the accompanying composition technique.

There are several common causes of deployment failure. An *unresolved dependency* occurs when a component depends on another component which is not present on the target system. This happens when the developer has either not made the dependence relation explicit to the deployment system (possibly being unaware of it), or has consciously moved responsibility for satisfying the dependency to another party, e.g., the user. The effort to install missing dependencies manually or to write an installer that does this, is substantial. For instance, an application like the Mozilla browser, in a particular configuration, requires the presence of 21 third-party components. Large systems (like open source operating system distributions) may consist of thousands of components.

Even if the required component is present, it may be an *incompatible version or variant*, that is, one that does not interoperate with the requiring component. Typically, the component is an older version that has certain bugs or does not provide necessary functionality, or is a newer version that is not sufficiently backwards-compatible. Also, the component may have been built with configuration options that cause it to be incompatible.

Component interference occurs when the installation of one component interferes with the operation of some previously installed component. This happens, for instance, when the installation of a component overwrites the files of a previously installed component. This will break components that use the latter, unless the new one is entirely compatible. Clearly, this is not the case in general, making interferences likely to occur.

From these examples we can distill two basic problems. The first problem is to **prevent unresolved component dependencies**. When deploying a component it is necessary to also deploy all components used by it. To that end, we need to identify component dependencies. In existing package managers, these have to be specified manually, and there is a substantial risk that we miss certain dependencies. For example, the Red Hat Package Manager (RPM) [1] requires the developer to specify for each package upon what other packages it depends, e.g., that a package requires at least version 2.3 of the package `glibc`. What if we forget to specify such a dependency? If the target system already has this package, then the component will work anyway. It is therefore hard to validate that the dependency information is complete. Also, version numbers are unreliable: the assumption that *any* version greater than 2.3 works may well turn out to be wrong. In addition, *timeline issues* have to be taken into account. The build of a component generally requires different components than the use of it. These must be identified separately. RPM for instance allows the separate specification of run-time and build-time dependencies, creating more opportunities for mistakes. These points in time are not unrelated. For example, Unix libraries that are *statically* linked into an application are exclusively build-time dependencies, while if they are *dynamically* linked—often just a difference of one flag in the build process—then they are also run-time dependencies.

The second problem is to **prevent component interference**. As described above, different components can interfere with each other, e.g., the upgrade of a component overwriting an older version. We then essentially have two similar but different components that both occupy the same locations in the file system. Thus, installing one destroys the other, possibly along with the consistency of the software that depends on it. This problem is particularly present in Unix systems, with their reliance on 'standard' paths, such as `/usr/bin`—there can be only one component stored at `/usr/bin/gcc`, so if two builds require different versions of that file, we are in trouble. The problem applies to Windows as well, e.g., conflicting versions of libraries in

`C:\Windows\System32`.

In conclusion, we need a way to (i) reliably identify component dependencies, and to (ii) prevent component interference. The remainder of this paper describes a solution to these problems.

3. Viewing the file system as program memory

In this section we recast the deployment problems identified in the previous section in terms of concepts from the domain of memory management in programming languages. Where programs manipulate memory cells, deployment operations manipulate the file system. This analogy reveals that safeguards against abuse of memory applied by programming languages are absent in deployment.

Components, as we defined them, exist in a file system and can be accessed through paths, sequences of file names that specify a traversal through the directory hierarchy, such as `/usr/bin/gcc`. We can view a path as an *address*. Then a string representing a path is a *pointer*, and accessing a file through a path is a pointer *dereference*. Thus, component interference due to file overwriting can be viewed as an address collision problem: two components occupy overlapping parts of the address space.

Furthermore, we can view components as representations of *values* or *objects*. Just as objects in a programming language can have references to other objects, so can components have references to other components. If dereferencing a pointer is not possible because a file does not exist, we have the deployment equivalent of a *dangling pointer*. A component A is dependent on a component B if the set of files that constitutes A *enables* an execution involving A to dereference pointers to the files constituting B. For example, an application that is dynamically linked against a file `/lib/libc.so` (a C library) enables a pointer to that file since execution of the application will cause a dereference of that file when it is started. If, however, the file is missing, the deployed application contains a dangling pointer. We say "enables an execution" because A itself might not be executed or even be executable; e.g., when A is a configuration file containing a path to B, and this file is read by some other component C.

To prevent dangling pointers, we should consider the various ways through which a component A can enable a pointer to another component B. Since components are similar to objects, we can provide analogues to the ways in which a method of a class can obtain a pointer to another object. First, it may be that a pointer to B was passed to the build process that constructed A. For example, this is often the case for Unix-style dynamically linked libraries: the build of an application stores the full path to the library in the application binary. This is equivalent to the constructor of an object storing a pointer that was passed in:

```
class Foo {
  Bar x;
  Foo(Bar y) { x = y; }
  int run() { return x.doIt(); }
}
```

Here, the execution of the constructor is similar to the construction of a component, and the execution of method `run()` is similar to the use of a component.

Note that on the other hand, a pointer passed in at build-time may also be completely "consumed" and therefore can no longer cause a dangling pointer. Examples include pointers to static libraries, or the compiler, which are not usually retained in the result. This is comparable to a constructor that uses a pointer to another object to compute some derived value but does not store the pointer itself:

```
class Foo {
  int x;
  Foo(Bar y) { x = y.doIt(); }
  int run() { return x; }
}
```

The second possibility is late binding, that is, the pointer is not passed in when it is built, but rather when it is executed. This may happen through environment variables (such as the `PATH` program search path on Windows and Unix), program arguments, function arguments (in the case of a library), registry settings, user interaction, and so on. Conceptually, this is similar to:

```
class Foo {
  Foo() {}
  int run(Bar y) { return y.doIt(); }
}
```

Finally, and orthogonal to the previous methods, a pointer to B can be obtained using *pointer arithmetic*. Since paths are represented as strings, any form of string manipulation such as concatenation may be used to obtain new paths. If we have a pointer to `/usr/bin`, than we may concatenate the string `gcc` to obtain `/usr/bin/gcc`. Note that the names to be appended may be obtained by dereferencing directories, that is, reading their contents.

It follows from the above that it is hard to find the set of pointers enabled by a component. First, pointers may already be present in the source. Second, pointers passed in at build-time may or may not be stored in the component. Obviously, we do not want to distribute the compiler along with an application just because it was used to build it—but other build-time components, such as dynamic libraries, should be. Finally, pointer arithmetic can be used to obtain new pointers in uncontrolled ways. For correct software deployment it is essential that no dangling pointers can occur. Thus, we need a method to detect these.

4. File system closures

As noted above, dangling pointers in components are a root cause of deployment failure. Thus, to ensure successful software deployment, we must copy to the target system not just the files that make up the component, but also all files to which it enables pointers. Formally, the set of files to be included in the distribution of a component is the *closure* of the set of files in the component under the enables-a-pointer-to relationship. A file is characterised as a tuple (p, c) where p is the path and c is the contents required at that path. The contents of a path include not just file contents if p is a regular file, but also metadata such as access permissions. In addition, if p is a directory, the contents include a mapping from directory entry names to the contents of these directory entries. Hence, the closure of a set of files C is the smallest set $C' \supseteq C$ satisfying

$$\forall (p, c) \in C' : \forall p_{ref} \in \mathrm{Ptrs}(c) : \exists (p', c') \in C' : p_{ref} = p'$$

where $\mathrm{Ptrs}(c)$ denotes the set of pointers enabled by the contents of c. That is, if a path p is in the closure, then the paths to which it enables pointers must also be.

By definition, a closure does not contain dangling pointers. A closure can therefore be distributed correctly to and deployed on another system. However, as we have seen in Section 3, determining the set $\mathrm{Ptrs}(c)$ is not generally possible. In the next section we provide a systematic method to determine this set.

5. A pointer discipline

In the previous section we saw that correct deployment without dangling pointers can be achieved by deploying file system closures. Unfortunately, determining the complete set of pointers is problematic.

In the domain of programming languages, we see the same problem. Languages such as C and C++ allow arbitrary pointer arithmetic (adding or subtracting integers to pointers, or casting between integers and pointers). In addition, compilers for these languages do not generally emit run-time information regarding record and stack layouts that would enable one to determine the full pointer graph. For example, this makes it impossible to implement garbage collectors that can precisely distinguish between garbage and live objects.

Other languages address this problem by imposing a *pointer discipline*: programs cannot manipulate pointers arbitrarily. For example, Java does not permit casting between integers and pointers, or direct pointer arithmetic. Along with run-time information of memory layouts, this enables precise determination of the pointer graph.

For file systems the problem is that arbitrary pointer arithmetic is allowed and that we do not know where pointers are stored in files. Certainly, if we restrict ourselves to components consisting of certain kinds of files, such as Java class files, we can determine at least part of the pointer graph statically, for instance by looking at the classes imported by a Java class file. However, this information would still be incomplete due to the possibility of dynamic class loading. Since the dependency information cannot be guaranteed to be complete and because this technique is not generally applicable, this approach does not suffice.

The solution to proper dependency identification comes from *conservative garbage collection* [2]. This is a technique to provide garbage collection for languages that have pointer arithmetic and no run-time memory layout information, such as C and C++. Conservative garbage collection works by constructing a pointer graph during a "mark phase" by assuming that anything that looks like a valid pointer, is a valid pointer.. During the "sweep" phase, any block of memory to which no pointer was found is freed.

Since there is a correspondence between objects in memory and files in a file system, we can borrow from conservative garbage collection techniques. From the "mark" phase, we borrow the technique of scanning for things that look like pointers. In Section 7 we show how we also borrow techniques from the "sweep" phase to implement garbage collection of files. However, these techniques are not applicable in a naive way because pointers are strings, and simply scanning for strings would yield too many false positives (e.g., the fact that the strings `usr`, `bin` and `gcc` occur in a component, does not necessarily imply that the component is dependent on `/usr/bin/gcc`).

The solution is to *shape* pointers in such a way that they do become reliably recognisable. We do this by including in the path of a component a long, distinguishing string, e.g.,

```
/store/acbd18db4cc2f85cedef654fccc4a4d8-foo/
```

might be the path to some component `foo`. We can now easily find pointers by scanning files for occurrences of the hexadecimal part of the path, which acts as a unique identifier. For instance, suppose that we have a component `bar`,

```
/store/37b51d194a7513e45b56f6524f2d51f2-bar/
```

we can determine if `bar` depends on `foo` by *scanning* the file system objects under `bar` for the string `acbd...a4d8` (which may appear in the contents of a regular file, as the target of a symbolic link, or even as part of a file name). If we find the hexadecimal string part of the pointer, we safely conclude that `bar` is likely to be dependent on `foo`; otherwise, we assume that no such dependency relation exists.

As noted in Section 3, the build of a component may retain a pointer to another component, thus propagating a dependency to later points on the deployment timeline. In conventional deployment systems, these dependencies have to be specified separately, with all the risks inherent in manual specification. By analysing those files of a component that are part of a distribution (in source or binary form) or of an installation, we can automatically detect timeline dependencies, such as build-time and run-time dependencies.

What are the risks of our approach? They are the same as for conservative garbage collection. *Pointer hiding* occurs when a pointer is encoded in such a way that it is not recognised by the scanner (a false negative). For example, our implementation (see Section 7) assumes that the paths are stored as plain ASCII strings; if they were encoded in, say, UTF-16, or contained in compressed executables, our scanner would not recognise them. Such false negatives may cause a dependency analysis to produce incomplete results. However, as yet, our naive scanner has never missed a *single* dependency. If needed, the scanner can always be extended to recognise common representations.

6. Persistence

The technique of pointer scanning, as described in the previous section, solves our first problem of reliable identification of component dependencies. Below we describe a solution for the second problem of component interference, which occurs when two components occupy the same addresses in the file system. When we deploy software by copying a file system closure to another system, we run the risk of overwriting other software. The underlying problem is similar to that in the notion of *persistence* in programming languages, which is essentially the migration of data from one address space to another (such as a later invocation of a process). We cannot simply dump the data of one address space and reload them at the same addresses of the other address space, since those may already be occupied (or may be invalid). This problem is solved by *serialising* the data, that is, by writing it to disk in a format that abstracts over memory locations.

Unfortunately, we cannot apply an analogue of serialisation to software deployment, because it requires changing pointers, which cannot be done reliably in files. For instance, a tempting approach would be to rename the files in a closure to addresses not existing on the target system. To do this, we would also have to change the corresponding pointers in the files. However, patching files in such a manner is unlikely to work in general, e.g., due to internal checksums on files being invalidated in the process.

Thus, we should choose addresses in such a way as to minimise the chance of address collision. To make our scanning approach work, we already need long, recognisable elements in these file names, such as hexadecimal representations of 128-bit values as suggested above. Not every selection of values prevents collisions: e.g., using a combination of the name and version number of a component is insufficient, since there frequently are incompatible instances even for the same version of a component.

Another approach is to select random addresses every time we build a component. This works, but it is extremely inefficient; components that are functionally equal would obtain different addresses on every build, and therefore might be stored many times on the same system. That is, there is a complete lack of *sharing*: equal components are not stored at the same address.

Hence, we observe a tension between the desire for sharing on the one hand, and the avoidance of collision on the other hand. The solution is another notion from memory management. *Maximal sharing* [4] is the property of a storage system that two values occupy the same address in memory, if and only if they are equal (under some notion of equality). This minimises memory usage in most cases.

The notion of maximal sharing is also applicable to deployment. We define two components to be equal if and only if the inputs to their builds are equal. The inputs of a build include any file system addresses passed to it, and aspects like the processor and operating system on which it is performed. We can then use a *cryptographic hash* [13] of these inputs as the recognisable part of the file name of a component. Cryptographic hashes are used because they have good collision resistance, making the chance of a collision negligible. In essence, hashes thus act as unique "product codes" for components. This is somewhat similar to global unique identifiers in COM [3], except that these apply to interfaces, not implementations, and are not computed deterministically. In summary, this approach solves the problem of component interference at local sites *and* between sites by imposing a single global address space on components. The implementation of this approach is discussed in the next section.

7. Implementation

We have implemented our ideas in a build and deployment system called *Nix*[1]. The architecture is outlined in Figure 1. At the bottom of the system is a designated part of the file system called the *store*. This is where all components (including sources) live. The names of directory entries in the store contain hexadecimal representations of 128-bit MD5 [13] hashes of the inputs involved in building them. The computation of components from other components is described by *Nix expressions*, discussed below. The Nix tool manages the store on the basis of these expressions. Because Nix expressions describe completely static configurations of components in terms of hash codes, they are hard to write by hand. Therefore, they are generally *instantiated* from higher level *Fix expressions* using a tool

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

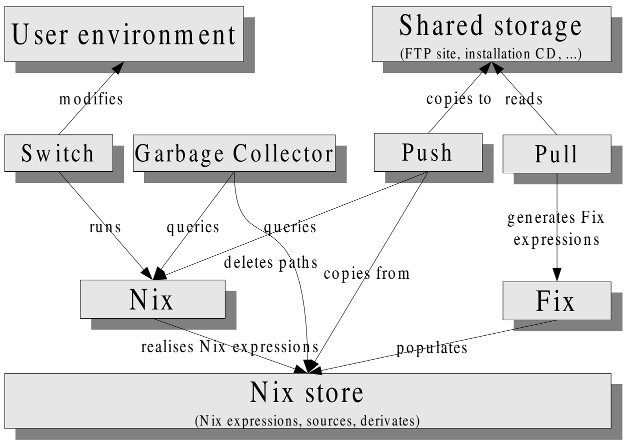

Figure 1. The Nix architecture

called *Fix*. Finally, several operations are defined using the Nix and Fix primitives, such as a *garbage collector* that automatically deletes unreferenced paths, a *switch* operation for maintaining user software environments, and *push* and *pull* operations for deployment.

Nix expressions. Nix expressions describe file system closures, or the computation of these. A *closure expression* consists of a set of *closure elements*. An element is a path in the closure along with the set of paths in the closure to which it contains pointers. One of the paths in the closure is the *root path*, the purpose of which becomes apparent below. Thus, a closure expression defines a rooted pointer graph, denoted as **Closure**`(`*root*`, [(`*pathname*`, [`*pointer*`])])`. (Here, square brackets denote sets, and parentheses denote tuples.) A *derivation expression* describes the computation of a closure from closures described by other Nix expressions (called *inputs*), such as sources, build tools, and libraries. It contains the following three pieces of information. First, it specifies the path that will be built by the derivation. Second, it lists the paths to Nix expressions for its inputs, together with symbolic names for later reference during the build process. The mandatory input named `builder` denotes the program to be executed to perform the build. Third, it lists the platform (processor and operating system) on which the builder is to be executed. Derivation expressions are denoted as **Derive**`(`*pathname*`, [(`*tag*`, `*pathname*`)], `*platform*`)`.

It should be noted that neither closure nor derivation expressions refer to the *contents* of a path, in apparent contrast to the definition of closures in Section 4. However, we assume that builders are pure functions (i.e., given the same inputs, they always build the same output). Since the name of a path contains a cryptographic hash of all inputs involved in building a path, the contents of a path are in

Realise(e):
  if e is of the form **Closure**(*root*, *elems*):
    for each (p, *ptrs*) $\in$ *elems*:
      if p does not exist:
        if there exists a substitute e' for p:
          Realise(e')
        else: abort
    return e
  else: e is of the form **Derive**(*out*, *ins*, *platform*)
    if there exists a successor e' of e: **return Realise**(e')
    Check whether the current platform satisfies *platform*
    env **:=** $\emptyset$
    elems **:=** $\emptyset$
    for each (*tag*, *path*) $\in$ *ins*
      e_{in} **:= Realise**(expression stored at *path*)
      env **:=** *env* $\cup$ (*tag*, root path of e_{in})
      elems **:=** *elems* $\cup$ the closure elements of e_{in}
    Execute *env*[**"builder"**] with environment env
    ptrs **:=** the paths in *elems* referenced in *out*
    elems' **:=** closure of set `[(`*out*, *ptrs*`)]` in *elems*
    e' **:= Closure**(*out*, *elems'*)
    Store e' and register it as the successor of e
    return e'

Figure 2. Expression realisation

essence encoded into a path name! Nix cannot enforce or verify that builders are pure. A common “minor” source of impurity is storing the system time in the output, but this generally does not affect the behaviour of a component.

The fundamental operation on a Nix expression is *realisation*, shown in pseudocode in Figure 2. (Our implementation provides transactional semantics to ensure correctness in the face of system failure or concurrency; these details are omitted.) A closure expression is realised by ensuring that the paths in the closure exist in the file system. A path that does not yet exist can be realised if a *substitute expression* for the path has been registered with the Nix system. Substitute expressions are Nix derivation expressions that obtain path contents from some installation source, such as an FTP site or a CD-ROM. This provides a flexible means for realising paths using a varying number of different access methods. If a substitute expression is not registered for a path, the realisation operation fails.

A derivation expression is realised by *normalising* it into a closure expression by performing the build action described by it. Thus, Nix expressions form a very simple calculus, with normalisation as the only reduction rule. To prevent successive normalisations of the same derivation, the closure expression e' resulting from a derivation expression e is registered as a *successor* in a persistent mapping. Hence, the algorithm first checks to see if a successor is already known. If not, we perform the build action after

Figure 3. Derivation expression for the Pan component

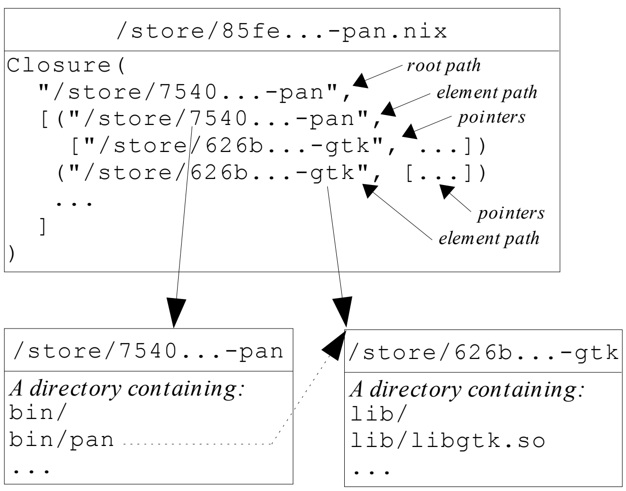

Figure 4. Closure expression resulting from normalisation of Figure 3

first realising all inputs. The root paths of the input closures are communicated to the builder through environment variables. If the build finishes successfully, we construct a closure expression rooted at the output path and containing all input elements that are reachable through pointers in the output path. We determine these by scanning the output path for the hash substrings of the paths of the input elements. This yields the resulting closure expression, which is placed in the store (in a path based on a hash of its value) and registered as a successor of the original expression.

An example of normalisation is shown in Figures 3 and 4. The first figure shows a simplified derivation expression for the Pan component, a newsreader for X11. The boxes represent paths in the Nix store, like objects on a memory heap, with the pointers between them. The Pan component depends on several inputs such as a build script, which is an atomic value identified by a simple closure expression, and the GTK component, a GUI toolkit which itself has many dependencies. Note the difference between expressions and the paths that they build: the derivation expression for Pan is stored in `9e47...-pan.nix`, while the resulting component is stored in `7540...-pan`. When the Pan derivation is normalised, the inputs are first realised. Closure expressions such as the build script are trivially realised, while derivation expressions such as the GTK component themselves have to be normalised. The builder unpacks the sources, builds them, and installs the result in the output path. Figure 4 shows the result of the normalisation. A closure expression describes the complete dependency graph of the Pan component. For example, the Pan executable has been dynamically linked to the GTK dynamic library, storing its full path (indicated as a dashed line). Thus, the Pan component has a static run-time dependency on GTK, preventing interference at run-time due to late binding—a very useful property for deployment. All such pointers are discovered by scanning the path just built. Note that any later normalisation of the Pan component or any of its subcomponents will finish in $O(1)$ time, since successors have been registered (e.g., from `9e47...` to `85fe...`).

Generating Nix expressions. It should be apparent from Figures 3 and 4 that Nix expressions are hard to write by hand. Indeed, due to the use of hashes, a change to any part of the derivation graph will propagate upward: modifying the source of a component will change not only the path of that component, but also those of all components that depend on it.

Fortunately, Nix expressions are not intended to be written by hand. Rather, they are *generated* from higher level component descriptions. Due to this layered architecture, many different generation policies are conceivable. The Nix system provides a generator called Fix. Figure 5 shows a Fix description for the Pan newsreader. Sources and Fix expressions for other components are referenced using ordinary file names. During instantiation, Fix will copy sources to the store and compute their hash codes. Referenced components are instantiated recursively. Unique hash codes for components are generated by Fix by creating a Nix ex-

```
Component(
  [ ("name", "pan-0.14.0.90")
  , ("builder", Relative("pan/pan-build.sh"))
  , ("src", Relative("pan-0.14.0.90.tar.bz2"))
  , ("gtk", Include("gtk+/gtk+.fix"))
... ])
```

Figure 5. Fix module for the Pan newsreader

pression with the output path omitted, computing a cryptographic hash over the text of that expression, and using that hash for the output path.

Fix is a simple functional language, allowing abstraction over component definitions. Arguments to functions can specify *bindings* for *variation points*, leading to the instantiation of different variants of a component. Using conditionals, variants can differ in their dependencies on other components. By calling such functions from other component descriptions with bindings for the variation points, concrete components are obtained. Note that instantiations with different arguments have different paths. Thus, variants automatically can coexist in a system.

Deployment. The Nix tool itself does not do deployment, but it provides the necessary *mechanism* to support various deployment *policies*. The tools `nix-push` and `nix-pull` in Figure 1 implement one possible deployment scheme. Defining additional deployment schemes is straightforward. Given a Nix expression, `nix-push` compresses and copies the expression and all referenced paths to some HTTP server. For a closure expression, the referenced paths are simply all the paths in the closure. For a derivation expression, it is the union of all paths referenced by the inputs. Note that the former typically performs a "binary" distribution, while the latter does a "source" distribution. The client performs a `nix-pull` to register substitute expressions for *all* the paths available on the server. These expressions define how paths can be downloaded from the server. That is, fetching the paths is done *lazily*: only when the client realises the expression, the paths are copied to the target system by realising the substitute expressions.

Component activation. Of course, it is not enough to be able to build and deploy components; ultimately application components must be *activated*, that is, made available to the end user. The set of activated packages is called the *configuration*. The method of activation is bound to the interface through which the application is to be accessed. For instance, activation might involve adding an entry to the Windows start menu, placing an icon on the desktop, or adding the application's executable to a directory in the PATH environment variable.

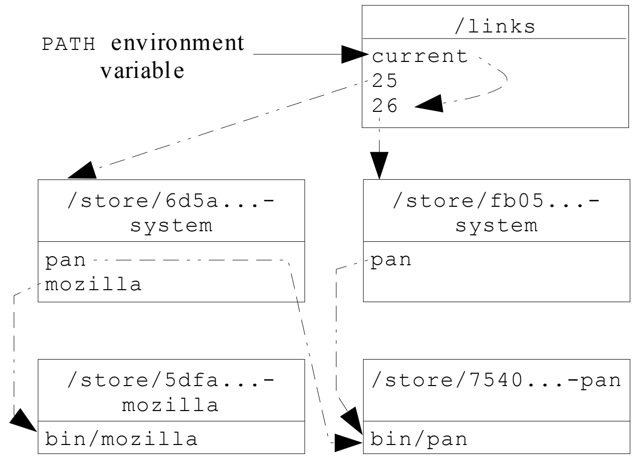

Figure 6. Application (de)activation

Figure 6 illustrates activation through the PATH variable. Trivial *system components* are generated automatically (using Fix) from a user specification of the desired applications. The output paths of these components consist of symbolic links (transparent aliases in the Unix file system, drawn as dashed lines) to the activated applications. Thus, if such a path appears in the user's PATH, the applications in it are in scope, e.g., running the command `pan` would start Pan. Rather than let the PATH variable point directly to such components in the store, we add two levels of indirection. First, we let PATH include the symbolic link `/links/current`, which points (indirectly) to the actual configuration. This prevents one from having to change the environment variable to change to a different configuration (so that, e.g., the user does not have to log out to have the new configuration take effect). Second, `current` itself points to a *configuration link*. New configurations are activated using a tool called `nix-switch`. This tool builds the new configuration, creates a link to its output path in the `/links` directory with a numerical name one higher than the previous configuration link (e.g., 25), and then changes `current` to point to the new configuration link (26). The latter step can be implemented as an atomic operation on Unix; thus, entire system configurations can be upgraded atomically. It is also possible to downgrade (or *roll-back*) by changing `current` to point back to a previous configuration link. A trivial extension to this scheme allows per-user (or even per-process) configurations.

Garbage collection. To ensure that no dangling pointers can occur, Nix does not provide an operation to *delete* components. Rather, paths are deleted from the store when they become *garbage*, i.e., when they are no longer reachable from outside the store. The `/links` directory serves as a set of roots for the collector. For example, in Figure 6, configuration 25 has a link to Mozilla, but configuration 26 does not (the user deactivated it). If configuration link 25 is deleted, the configuration becomes garbage and will be deleted when the garbage collector is run. Assuming that Mozilla is not reachable in some other way, it too will be deleted. Several policies are possible for running the collector, such as after every switch, when disk space becomes scarce, or from a scheduler.

8. Experience

The goals of this research are to prevent component interference and to enable reliable dependency determination. We obtained experience with Nix by creating Nix packages for 85 third-party components (ranging from simple components such as Unzip to complex ones such as Mozilla). Non-interference in the file system is assured through the use of cryptographic hashes of all build inputs in paths; the probability of a hash collision is extremely low. Indeed, using different versions and variants of these components in parallel presented no conflicts, e.g., there was no interference between different versions of shared libraries.

The question of whether we have attained more reliable dependency determination requires closer scrutiny. Nix components cannot obtain undeclared references to other components in the store, since that would require guessing a 128-bit hash code[2]. However, they *can* use components stored outside the store. This creates a risk of undeclared dependencies; e.g., if a component invokes a program such as `/bin/sh`, we cannot detect this. This is a basic limitation of our approach: pointer scanning only works on pointers with hash components—that is why we use them, after all. In a "pure" Nix-based environment, where all components are in the store, this problem does not occur.

However, the risk of "contamination" can be mitigated in several ways. For instance, our Nix packages include a bootstrapped build environment, i.e., a C compiler, linker, X11 client libraries, and so on. The C compiler and linker were configured not to search standard header file and library locations, substantially decreasing the risk of contamination. To measure the effectiveness of this approach, we compared 47 Nix packages to the corresponding packages of two other deployment systems: the FreeBSD Ports Collection[3] and Gentoo Linux[4]. The Nix packages were specified independently, without reference to any other package management system. They were built on a fairly standard SuSE Linux 8.2 system containing 573 SuSE packages. Thus, the risk of contamination was substantial. However, we found *no* missing dependencies in any of the Nix packages (apart from optional dependencies) using this method, an indication of the success of our approach in preventing undeclared dependencies. In contrast, we found several problems with the dependency specifications in Gentoo and FreeBSD, e.g., Gentoo's Pan package failed to declare a dependency on the `pkgconfig` package, Gentoo packages frequently (but not always) omitted dependencies on system packages such as Perl or the C library, and several FreeBSD packages omitted a dependency on Perl.

[2] A component could search through the store directory. However, the use of hashes is not a security feature; it is a way to prevent *accidental* use of undeclared components. Furthermore, on Unix systems we can prevent this by removing read permission from the store directory.

[3] `http://www.freebsd.org/`

[4] `http://www.gentoo.org/`

9. Related work

Software deployment is often performed with package managers, such as RPM [1]. As motivated earlier, these suffer from several shortcomings which prevent correct deployment: i) dependency analysis is weak or missing; ii) dependencies are based on (ad-hoc) fragile version schemes; iii) there is usually no support for concurrent deployment of multiple versions and variants; iv) at best, timeline dependencies are supported only marginally.

Vesta [8] provides exact dependency determination for build management by performing builds on a virtual file system, thus allowing it to intercept all file system operation done by build tools. However, Vesta is limited to build-time dependency analysis; once components leave the Vesta environment (i.e., when they are deployed), there exist no means to determine remaining component dependencies. In addition, the approach is not portable.

In contrast to Nix, Autoconf[5] performs *dynamic* dependency resolution, which is performed at build-time by scanning the file system for installed components that are needed. We believe that this approach is flawed because: i) it gives rise to version and variant conflicts because dependencies are not precise; ii) it makes deployed software fragile because dependencies are not globally managed and new dependencies can be added implicitly; iii) it tends to place the responsibility for selecting the right prerequisite components on the installing party rather than the developer or distributor. Different (possibly incorrect) build results, or even build failures are an often seen consequence of this.

Software deployment and software development can be combined by integrating deployment systems with SCM systems [9, 14, 7]. Although currently not supported, we have plans to add this functionality based on our component identification mechanism.

Software deployment is often not explictly addressed as part of CBSE. In [11, 12] the problem of independently deployable components is addressed. They discuss policies for configuration, release, and version management. There

[5] `http://www.gnu.org/software/autoconf/`

is no mechanism for tracking dependencies but to install all available components in a predefined location.

The analogy between packages and components is addressed in [16], where they motivate that package management can be improved by elevating packages to components, as we do. Techniques for deployment of such components are discussed in [10, 6]. These techniques produce file system closures, but dependency analysis does not go as deep as in our approach. Furthermore, this analysis is performed manually and is therefore error prone. Finally, maximal sharing between applications is not supported which leads to significant redundancy of deployed components.

10. Conclusion and future work

In this paper we have presented a new discipline for software deployment inspired by memory management in programming languages. By computing product codes (hashes) that exactly identify components, we create a global address space for components, which allows complete transfer or reproduction of component installations between sites. The use of these product codes allows detection of dangling pointers and avoids component interference, while allowing concurrent installation of variants. Putting deployment components under the control of this discipline elevates them to the status of program objects under the control of a program, thus elevating deployment from a black art into a programming paradigm, providing complete programmable control over the deployment process.

The discipline and its implementation in the Nix deployment system provide a foundation for the implementation of a wide variety of software deployment and application management scenarios. Deployment activities such as deinstallation and upgrading follow naturally from the basic operations. But also more complex operations are in reach. We are currently working on a number of higher-level applications and extensions of Nix.

A *daily build system* regularly builds software products under development in various configurations, and typically has to deal with various baselines and multiple versions of a product. The use of Nix promises to make maintenance of dependencies in such as system tractable.

There is much room for improvement in the Fix language, notably with respect to the handling of *variability* in components: factoring out all variation points of small scale components leads to a explosion in the number of variation points of composite components. Also needed is an end-user *querying* mechanism that informs a user about the availability of component variants. This requires storing and presenting configuration information in a high-level manner. Component *state* is an important cause of deployment failure that must be addressed. Finally, we expect that the product code scheme can be extended to support distributed components.

Acknowledgments We thank Martin Bravenboer, Dave Clarke, and Armijn Hemel for their feedback on Nix and this paper.

References

[1] E. C. Bailey. *Maximum RPM*. Red Hat Press, 2000.

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

[3] D. Box. *Essential COM*. Addison-Wesley, 1998.

[4] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software—Practice and Experience*, 30:259–291, 2000.

[5] A. Carzaniga et al. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, University of Colorado, April 1998.

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

[7] M. de Jonge. Package-based software development. In *Proceedings of the 29th Euromicro Conference*, pages 76–85. IEEE Computer Society Press, Sept. 2003.

[8] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Technical Report Research Report 168, Compaq Systems Research Center, March 2001.

[9] A. van der Hoek. Integrating configuration management and software deployment. In *Proc. Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001)*, December 2001.

[10] A. van der Hoek and A. Wolf. Software release management for component-based software. *Software – Practice and Experience*, 33(1):77–98, Jan. 2003.

[11] R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.

[12] R. van Ommering. Techniques for independent deployment to build product populations. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

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

[14] S. Sowrirajan and A. van der Hoek. Managing the evolution of distributed and interrelated components. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management*, volume 2649 of *LNCS*, pages 217–230. Springer-Verlag, 2003.

[15] C. Szyperski. Component technology—what, where, and how? In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, May 2003.

[16] D. B. Tucker and S. Krishnamurthi. Applying module system research to package management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.