Concurrent Variants, Isolation, and Feature Models in Software Deployment

Eelco Dolstra

Center for Software Technology

Utrecht University

eelco@cs.uu.nl

February 27, 2003

Abstract

Software deployment has long been a neglected topic within the field of Software Configuration Management. Deployment is viewed as a trivial last step consisting of copying files to the right locations. However, many significant problems—the "DLL hell", package version conflicts, upgrade issues, validation drift, and even the recent Slammer worm—are the result of a undisciplined and *ad hoc* approach to deployment and package management. In this talk I will show some of the issues in deployment and discuss some better approaches such as Maak and Nix, as well the relationship to variability and feature models.

Overview

- Issues in software deployment & package management.
- Maak.
- Nix.

The Problem

The goal: to get software working on the intended target machines.
The units of deployment are called *packages*.
What do we want?

- Install packages without breaking other stuff.
- Uninstall packages.
- Upgrade packages.
- Maintain multiple concurrent variants (e.g., different versions, or builds with different feature selections of the same package).

Installation strategy: Old School Unix

Throw everything in a few system directories (`/bin`, `/usr/bin`, `/usr/share/man/man1`, `C:/Windows/System32`, etc.).

Problem: *unmanageable*.

- How do we remove a package?
- How do we install multiple concurrent versions/variants of a package?
- etc.

```
[eelco@vinkel:~]$ ll /usr/bin/ | wc -l
   1866
```

Quick Fix: Add a Database

E.g., RPM, FreeBSD packages, Debian `dpkg`.

For each installed package, remember in a database which files belong to the package.

This is a hack to make a bad strategy work. Note: a file system is a type of database; you just have to use it properly.

Installation strategy: separate directories

Put everything in a separate directory. Now many actions become quite simple:

- Removing a package (except for dependency issues!).
- Querying: which files belong to a package, or to which package does a file belong.
- Concurrent versions/variants (just put 'em in separate directories).

E.g., GNU Stow.

Example

```
/sw/pkg/emacs-21.2.1
/sw/pkg/emacs-21.2.1/bin
/sw/pkg/emacs-21.2.1/lib
/sw/pkg/emacs-21.2.1/...
/sw/pkg/gcc-2.95.3/...
/sw/pkg/gcc-3.2.2/...
```

And for convenience we can setup links in `/sw/bin` etc.:

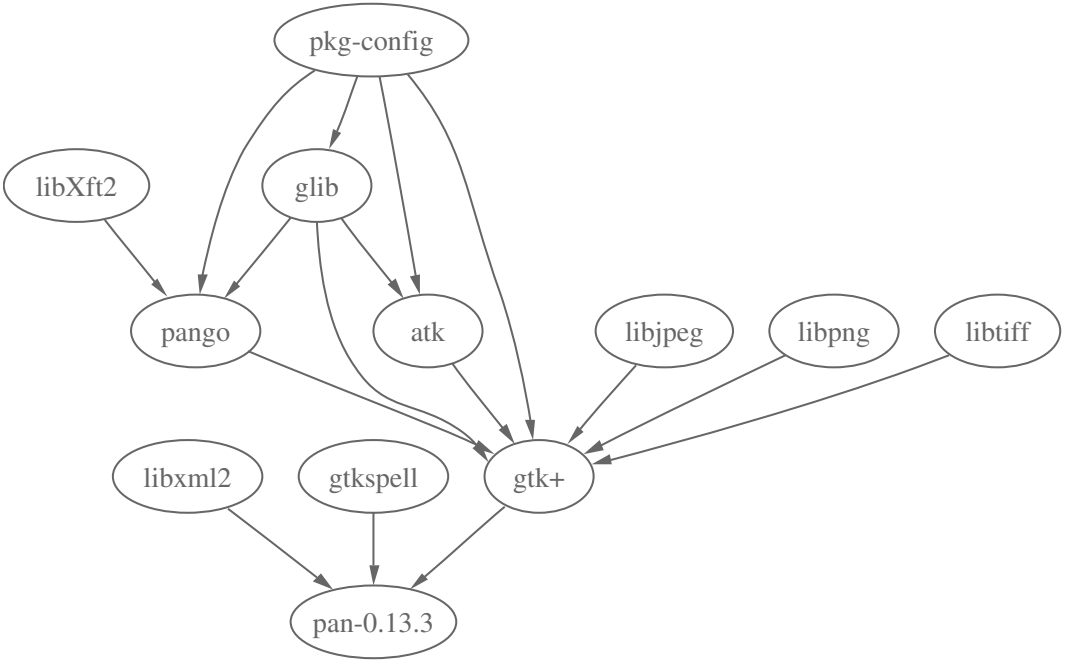
```
/sw/bin/gcc -> /sw/pkg/gcc-2.95.3/bin/gcc
/sw/bin/emacs -> /sw/pkg/emacs-21.2.1/bin/emacs
```

Problem: Many Dependencies

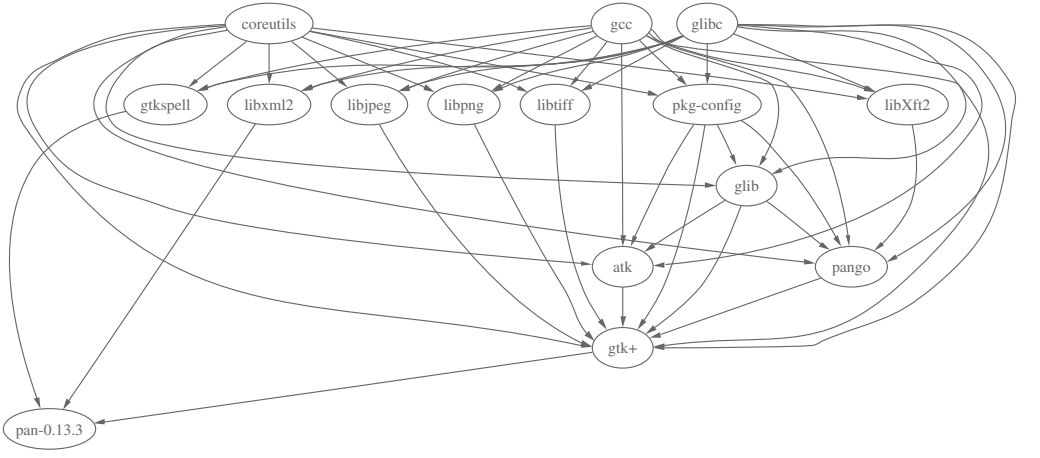
Packages can have many (indirect) *dependencies*.

This is especially a problem in the open source world, where there are typically many small packages that depend on each other.

Example


pkg-config
libXft2
glib
pango
atk
libjpeg
libpng
libtiff
libxml2
gtkspell
gtk+
pan-0.13.3

Example


coreutils
gcc
glibc
gtkspell
libxml2
libjpeg
libpng
libtiff
pkg-config
libXft2
glib
atk
pango
gtk+
pan-0.13.3

Problem: Source vs. Binary Distribution

- Mechanism and interface of source installation is completely different from binary installation.
- This is not so nice. Ideally, binary distribution is transparent; after all, a binary distribution can be considered conceptually to be a source distribution *that has been partially evaluated with respect to a target platform*.

Problem: Source vs. Binary Distribution (2)

- Dichotomy between build formalism and packaging formalism.
  - The issues in build and package management are partially overlapping. E.g., they both deal with dependencies among components.
- Research goal: how to merge build management and deployment / package management.

Example

Consider this Makefile:

```
program: main.o libfoo.a
    gcc -o program main.o libfoo.a

libfoo.a: foo.o bar.o
    ...
```

If we decide to make `libfoo` separately deployable, i.e., put it in a separate package, we typically end up with:

```
program: main.o
    gcc -o program main.o -lfoo
```

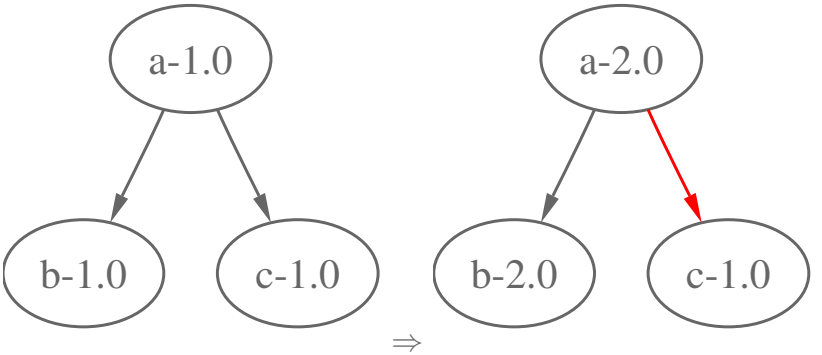
and we move the dependency to a higher level, say, an RPM specfile.

Problems:

- The dependency between `program` and `libfoo` is broken. Building the first won't trigger (re)building the second.
- `-lfoo` is an *uncontrolled input*. Where does the linker find `libfoo`? Is that actually the `libfoo` that we want? Etc.
- We now need two formalisms: the build specification and the package specification.

Problem: Upgrading

- Upgrades may need to be *atomic*.
- We should have the capability to *rollback*.
- Upgrades tend to be unsafe:



Example: the *main* cause of the Code Red / Slammer worm debacles was the difficulty in getting patches to the servers. Patching was:

1. Not automatic.
2. Risky.

Problem: Lack of Isolation

- *Lack of isolation*: package builds are not isolated from each other.
- This leads to *implicit dependencies*: package A depends on package B without declaring that it does so.
- This causes *irreproducibility*: packages that build in a certain environment may unexpectedly fail to build or be build incorrectly in another environment.

Maak

See also: E. Dolstra. Integrating Software Construction and Software Deployment. In *11th International Workshop on Software Configuration Management (SCM-11)*, Portland, Oregon, USA, May 2003. To appear.

Maak

Maak is a build tool, but its module system allows us to do deployment as well. The main features:

- Simple functional input language; makes *variant builds* easy.
- Files are values, tools are functions that produce other files/values; makes creation of variants easy.
- *Derivate tracing*; allows generic operations (`clean`, `dist`).
- Build auditing: verify that all inputs and outputs of an action are declared.
- Module system allows user-definable package management strategies.

Example

Simple compilation:

```
import stdlibs;

default = put (./test, link (./test.c));
```

(This is equivalent to:

```
default = x;
x = put (./test, y);
y = link (./test.c);
```

I.e., we have referential transparency.)

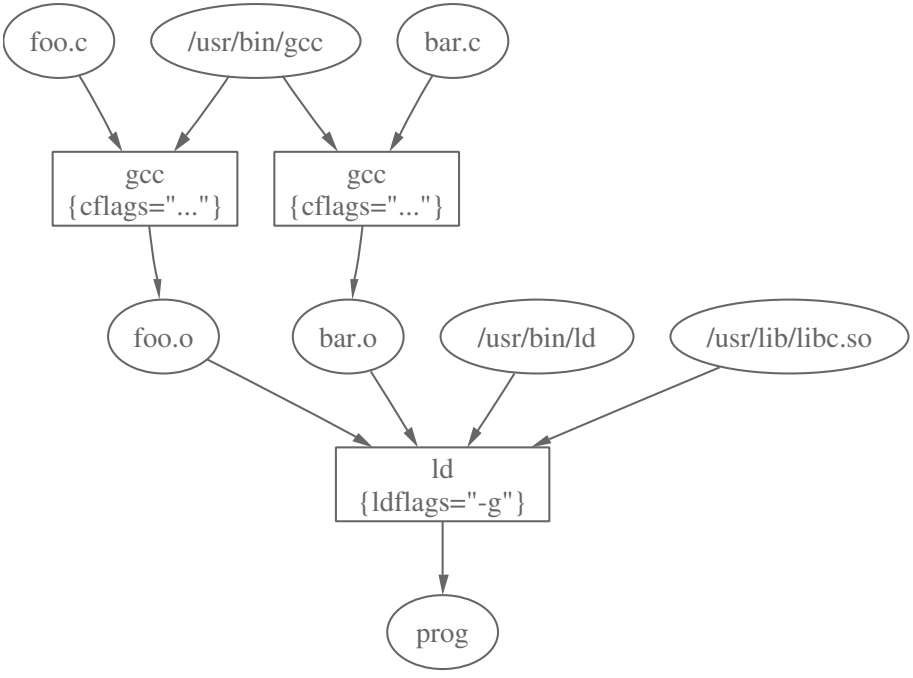
Model

The command `maak` *`target`* will evaluate variable *`target`*. The result should be a *build graph*. The nodes in this graph are:

- *Files*; they can either be *atoms* (irreproducible sources) or *derivates* (things to be created by tools). Derivates have an outgoing edge to an action.
- *Actions*. Actions are a collection of *attributes*. Attributes can be input files, output files, or basic values (strings, etc.).

  The special attribute `build` is a function that actually performs the build, usually by running external programs.

Example


foo.c
/usr/bin/gcc
bar.c
gcc
{cflags="..."}
gcc
{cflags="..."}
foo.o
bar.o
/usr/bin/ld
/usr/lib/libc.so
ld
{ldflags="-g"}
prog

Tool definition

We obtain tools simply by abstracting over actions. E.g.:

```
compileTest =
  { c = ./test.c
  , obj => ./test.o
  , build = exec "cc -c {c} -o {obj}"
  }.obj;
```

becomes:

```
compile = {c, obj}:
  { c = c
  , obj => obj
  , build = exec "cc -c {c} -o {obj}"
  }.obj;

compileTest = compile {c = ./test.c, obj = ./test.o};
```

(The syntax `{arg1, ..., argn}: expr` is just a function definition; it defines a function with arguments `arg1`, ..., `argn`).

Tool definition

A fancier compiler tool definition:

```
compileC = {in, cflags, includes, cc}:
  { in = in
  , cflags = cflags ? []
  , cc = cc ? /usr/bin/cc
  , includes = includes ? []
  , out => anonName (".o")
  , build = exec ("{cc} -c {in} {cflags} -o {out}")
  }.out;
```

Chaining

It is nice to have *chaining* of rules. In GNU Make this happens magically by chaining pattern rules, but that's bad because it's global.

Maak tries not to enforce a policy; chaining is left to the tool definitions. For example:

```
compileC = {in, cflags, includes, cc}:
  { in = forceC (in)
  , cflags = cflags ? []
  , cc = cc ? /usr/bin/cc
  , includes = includes ? []
  , out => anonName (".o")
  , build = exec ("{cc} -c {in} {cflags} -o {out}")
  }.out;

forceC = {x}:
  if (suffix (x) == ".c", x, bison (x));
```

Attribute propagation

Naive chaining breaks if we have to provide attributes to implicit actions. So we *propagate* attributes:

```
compileC = {in, cflags, includes, cc}:
  { in = forceC (args, in)
  , cflags = cflags ? []
  , cc = cc ? /usr/bin/cc
  , includes = includes ? []
  , out = derived ("o", this)
  , build = exec ("{cc} -c {in} {cflags} -o {out}")
  }.out;

forceC = {as, x}:
  if (suffix (x) == ".c", x, bison {as, in = x});
```

`args` is implicitly bound to all arguments of a function.

Attribute propagation: example

```
./test = link
  { in = [./test.c ./parser.y]
  , cflags = "-DDEBUG -g"
  , bisonflags = "--debug"
  };
```

The `cflags` and `includes` attributes are propagated to the `compileC` action.

The `bisonflags` attribute is propagated to the `compileC` action and then on to the `bison` action.

Variant builds

```
import stdlibs;
default = [libATerm libATerm-dbg ...];

srcs =
  [./aterm.c ./list.c (...) ./md5c.c];

libATerm =
  makeArchive {in = srcs});
libATerm-dbg =
  makeArchive {in = srcs, cflags = "-g"});
libATerm-ns =
  makeArchive {in = srcs, cflags = "-DNO_SHARING"});
....
```

Variant builds (2)

Even better:

```
atermLib = {debug, sharing}:
  makeLibrary
    { in = srcs
    , cflags = if (sharing, "", "-DNO_SHARING")
             + if (debug, "-g", "")
    };
```

Derivate file naming

Maak provides mechanism, not policy: tool definitions determine where derivates are stored. This is usually in a local `.maakcache` directory, but we could just as well use shared caches, store derivates in the same directory as the source (e.g., `test.c` $\Rightarrow$ `test.o`), etc.

Derivate file naming

In the default anonymous naming scheme, Maak generates a hashed name for files from the input attributes of the action that creates them.

For example: if `foo.c` is compiled in several variants, e.g., with `CFLAGS=""` and `CFLAGS="-g"`, we get two different files, e.g., `.maakcache/foo_5efd.o` and `.maakcache/foo_94ac.o`.

(If there is a hash collision, Maak will detect it; the build will still succeed, but there may be some work duplication due to clobbered targets).

This scheme has the benefit of implicitly *coalescing* equal (sub)values, e.g., in:

```
./test1 = link (compileC (./test.c));
./test2 = link (compileC {in = ./test.c, cc = /usr/bin/cc});
```

building `test2` will reuse the anonymous intermediates used in building `test1` since they have the same names and the same attributes. (`/usr/bin/cc` is the default for `cc`, so `test1` $==$ `test2`).

Module system

A Maakfile can import other Maakfiles:

```
import ./blah/Maakfile;
```

The argument to `import` is an arbitrary *expression* that should evaluate to a build graph rooted at a file node.

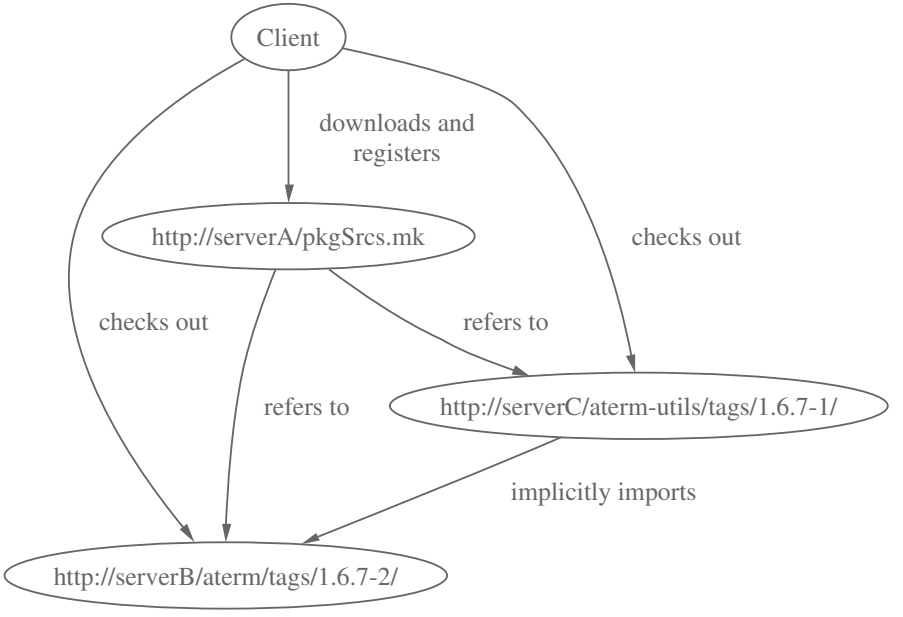
```
import stdlibs;  # defines pkg
import pkg("foo-1.2.3-1");
```

Example

Maakfile for package `aterm-utils`:

```
import stdlibs;
import pkg ("aterm-1.6.7-2");

default = progs;

progs = [termsize ...];

termsize = link' (./termsize, ./termsize.c);
...

link' = {out, in}: put (out, link
  { in = in
  , libs = [atermLib {debug = false, sharing = true}]
  , includes = [atermIncl]
  });

activate = map ({p}: activateExec (p), progs);
```

Example



So we can "install" (activate programs) in `aterm-utils` by running:

```
maak -f 'pkg ("aterm-utils-1.6.7-1")' activate
```

Principle 1: Exact dependency version

Package dependencies should almost always be *exact*, i.e., we should import a specific version of a package. So never:

```
import pkg ("aterm");
```

but instead

```
import pkg ("aterm-1.6.7");
```

(A list of versions that have been verified to work is also acceptable).

Rationale: builds tend to break with high probability if we have these "wildcard" dependencies. So none of that Autoconf stuff!

It's better to have 10 versions of package X on your working system than to have 1 version of package X on your non-working system.

Principle 2: No copying

- Don't install packages by copying.
- Copying tends to break stuff (e.g., due to dynamic linking).
- We *activate* packages by symlinking.

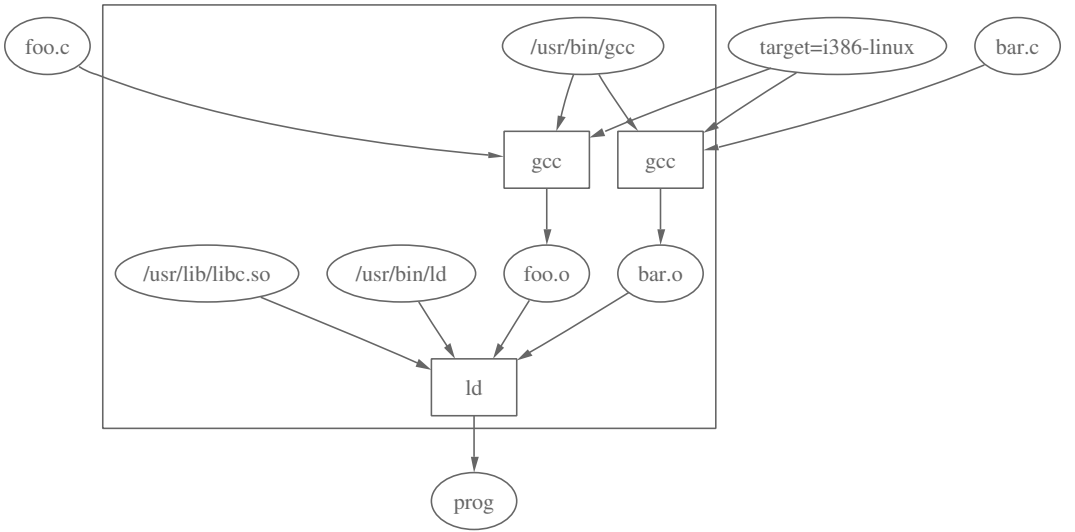
From Source to Binary Distribution

- We don't want to deploy binary packages explicitly. Instead, we just want to provide a link to the source along with a link to a suitably populated *cache of derivates*.
- This would, e.g., enable source-based OS distributions without the usual overhead (of compiling everything yourself): the system would transparently use pre-build derivates in the cache *if* the attributes match.

Barriers

Problem: how to prevent spurious recompilation due to minor changes in the environment (e.g., a different compiler)?

Solution: move certain dependencies into a subgraph.



Issues

- Isolation not enforced.
- Source $\longrightarrow$ binary distribution is still difficult; for example, we might not want to rebuild just because we're building in a different directory.
- Incompatible with existing build tools.

Nix

Nix

- Nix is a deployment and package management system (under development). Also a Linux-based operating system environment.
- It enforces isolation by randomizing package directories.
- I.e., there is no `/bin`, `/usr/lib`, etc.
- Package build scripts must explicitly import other packages.
- There is only *one* well-known file name: `/pkg/sys/bin/nix`, which provides the required component glue.

Package instantiations

- In order to support variants (including different versions, target platforms, etc.) we need to distinguish between *packages* and *package instantiations*.
- So we first have to tell the system where to find a package:

```
$ nix register-pkg aterm-1.6.7 http://...
```

- Then the act of *getting* a package will fetch the package if we don't already have it, run a script `prepare` within the package directory, and return its path:

```
$ nix get-pkg aterm-1.6.7
(wait...)
/pkg/aterm-1.6.7-fbe0aa536fc349cbdc451ff5970f9357
```

- So the path is randomized; packages cannot rely on other packages being in certain fixed locations. $\Rightarrow$ isolation
- There are other techniques to accomplish this, such as setting up a `chroot` environment for each build / execution (but that's more work).

Example

The prepare script of the aterm library:

```
#! /bin/sh

top=‘pwd‘

export PATH=$PATH\
  :‘nix get-pkg bash-2.05b-1‘/bin\
  :‘nix get-pkg coreutils-4.5.7-1‘/bin
  :‘nix get-pkg gmake-3.79.1-1‘/bin
export LIBRARY_PATH=‘nix get-pkg glibc-2.3.1-1‘/lib
export CC=‘nix get-pkg gcc-3.2.2-1‘/bin/gcc
export CFLAGS=" \
  -isystem ‘nix get-pkg glibc-2.3.1-1‘/include \
  -isystem ‘nix get-pkg kernel-headers-2.4.20-1‘/include"

cd src
./configure --prefix=$top
make
make install
```

From Source to Binary Distribution

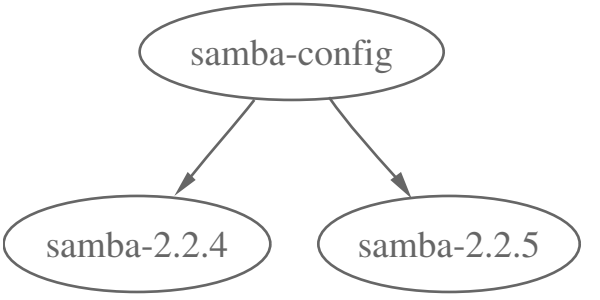
(Sketchy!)

- Just tar the package directory.
- Remember hash of input packages and attributes.

State packages

Packages have state; this state has to be maintained across upgrades.

Solution: factor out the state into a separate package that can be imported by the packages that need it.



Dependencies at other points on the timeline

- E.g., runtime execution dependency: program X from package A runs Y from package B. We can do this by invoking `nix` at runtime. For example, instead of

  ```
  mozilla http://foo/bar/
  ```

  we do

  ```
  ‘nix get-pkg mozilla-1.2-1‘/bin/mozilla http://foo/bar/
  ```

- This may cause a build operation at runtime.
- If this sort of demand building is undesirable it can be done in advance.

Feature models

- We need to annotate packages with a feature model to express possible variants.
- Make packages into functions: λ`args` $\rightarrow$ `instance`.
- Arguments can include references to other packages (i.e., late component composition).
- If each package has a feature model, we can show and manipulate them all together in one system wide configuration tool.

Conclusion and Future Work

- We need isolation between packages to ensure safe installs, build reproducibility, and so on.
- This is hard to guarantee in conventional systems.
- Maak basically works.
- Nix doesn't, and there are many open issues.