Integrating Software Construction and Software Deployment

Eelco Dolstra

Utrecht University

eelco@cs.uu.nl

May 9, 2003

Introduction

The software deployment process:

- Build.
- Package.
- Install.

So we have to deal with:

- The build system.
- The deployment system.

Problem #1: Overlap

Both deal with *dependencies between components*. Consider, e.g.,:

```
libATerm.a: aterm.o gc.o ...
    ...

termsize: termsize.o libATerm.a
  cc -o termsize termsize.o libATerm.a
```

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

```
termsize: termsize.o
  cc -o termsize termsize.o -lATerm
```

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

Problem #1: Overlap (cont'd)

The inter-component interface:

1. Fetch source code for package `aterm`.
2. Configure and build it.
3. Install it $\Rightarrow$ copy the library and C header files to, e.g., `/usr/lib`, `/usr/include`.
4. Fetch source code for package `aterm-utils`.
5. Configure and build it $\Rightarrow$ entails specifying or finding the location of the `aterm` library (Autoconf).
6. Install it $\Rightarrow$ copy the programs to, e.g., `/usr/bin`.

Problem #1: Overlap (cont'd)

Consequences:

- Installation becomes harder.
- Dependencies are hidden from the lower level. (E.g., Make no longer sees them $\Rightarrow$ updates don't propagate).
- Developing on compositions of separate components becomes harder, so fine-grained component reuse becomes less attractive.

Problem #2: Variability

- Large software systems typically have a very large number of potential instantiations. (Linux 2.4.20 has > 1500 variation points).
- We have to manage each deployable variant.

Problem #3: Explicit Packaging

- Ideally we view binary deployment as an optimisation of source deployment.
- This should happen transparently.

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 — Maakfile for `aterm`

```
import stdlibs;

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

atermInclude = ./;

srcs = [ ./aterm.c ./gc.c ... ];
```

Example — Maakfile for `aterm-utils`

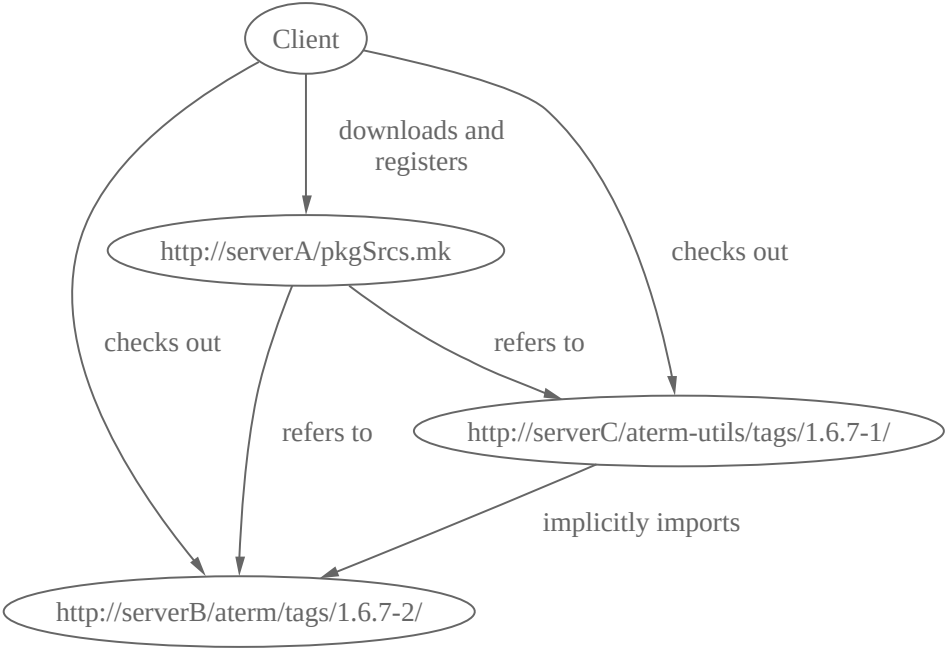
```
import stdlibs;
import pkg ("aterm-1.6.7-2");

default = [termsize ...];

termsize = link
  { in = ./termsize.c
  , libs = [atermLib {debug = false, sharing = true}]
  , includes = [atermIncl]
  };
```

A Deployment Strategy

The function `pkg` fetches packages from Subversion repositories.



A Deployment Strategy (cont'd)

The command

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

will recursively

- Fetch and build the required variant of the ATerm library.
- Fetch and build the ATerm utilities.

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 enables source-based OS distributions without the usual overhead (of compiling everything yourself): the system can transparently use pre-build derivates in the cache *if* the build attributes match.
- The *obfuscating properties* of binary distributions are accidental; can also be accomplished through *source-to-source transformations*.

Related Work

- Build management: Odin (Clemm), Vesta (Heydon et al.), Amake (Baalbergen), ...
- Package management: RPM, FreeBSD Ports Collection, ...
- M. de Jonge, *Source tree composition*.
- A. van der Hoek, *Integrating configuration management and software deployment* (CDSA 2001).

Conclusion

Integrating build management and deployment is important:

- Removes the discontinuity in formalisms/tools used for building and deployment.
- Simplifies handling of variability.
- We accomplish this through a module system that enables policy freeness w.r.t. component locations.
- Allows transparent source/binary distribution
- Removes the separation between SCM on the developer side and the client side.