Integrating Software Construction and Software Deployment

Eelco Dolstra

Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
eelco@cs.uu.nl

Abstract. Classically, software deployment is a process consisting of building the software, packaging it for distribution, and installing it at the target site. This approach has two problems. First, a package must be annotated with dependency information and other meta-data. This to some extent overlaps with component dependencies used in the build process. Second, the same source system can often be built into an often very large number of *variants*. The distributor must decide which element(s) of the variant space will be packaged, reducing the flexibility for the receiver of the package. In this paper we show how building and deployment can be integrated into a single formalism. We describe a build manager called *Maak* that can handle deployment through a sufficiently general module system. Through the sharing of generated files, a source distribution *transparently* turns into a binary distribution, removing the dichotomy between these two modes of deployment. In addition, the creation and deployment of variants becomes easy through the use of a simple functional language as the build formalism.

1 Introduction

Current SCM systems treat the building of software and the deployment of software as separate, orthogonal steps in the software lifecycle. Controlling the former is the domain of tools such as Make [1], while the latter is handled by, e.g., the RedHat Package Manager [2]. In fact, they are not orthogonal. In this paper we show how building and deployment can be integrated in an elegant way.

We shall first look at the problems inherent in the current approach to building and deployment.

Component dependencies Separating the building and deployment steps leads to a discontinuity in the formalisms used to express component dependencies. In a build manager it is necessary to express the dependencies between source components; in a package manager it is necessary to express the dependencies between binary components.

An example may be helpful. Consider the following Makefile for a small system that consists of a support library and a main program.

```
program: main.o libsupport.a
  cc -o program main.o libsupport.a

libsupport.a: foo.o bar.o
  ...
```

The Makefile properly expresses the relationships between the components; e.g., if any of the sources of the library changes, the program will be rebuilt as well.

On the other hand, if we were to take out the support library and move it into a separate package — so that it could be deployed independently — we end up with two Makefiles that do not provide a sufficient amount of dependency information, e.g., for the library package:

```
libsupport.a: foo.o bar.o
  ...
```

and for the program package:

```
program: main.o
  cc -o program main.o -lsupport
```

Note that the dependency of the program component on the library component is no longer explicit at the Makefile level. We now have to express the dependency at the package level, i.e., we have to express that the program package has a build-time or run-time dependency on the library package, in the case of static or dynamic linking, respectively. That is, splitting one package into several lifts dependencies to a higher level, making them invisible to the lower level.

Source vs. Binary Distribution Another issue is the dichotomy between source and binary distributions. In an open-source environment software is provided as source packages and sometimes as binary packages. The packaging and installation mechanisms for these are quite different. This is unfortunate; 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* using, e.g., a compiler.

Variability Most software systems exhibit some amount of *variability*, that is, several *variants* can be instantiated from the same source system. Typical *variation points* are choosing between exclusive alternatives, whether to include optional features, and so on.

In open-source systems in particular there tend to be many build-time variation points in order to make the system buildable in a wide variety of environments that may or may not have the characteristics necessary to implement certain features. In turn this is due to the use of fine-grained deployment strategies: many small independent packages are used to compose the system. For example, a program might have the ability to generate graphical output in JPG

and PNG format, but only if the respective libraries `libjpg` and `libpng` are available.

Open-source distributors typically need to use binary packages for speed of installation. Unfortunately, binary packages tend to force a "one-size-fits-all" approach upon the distributors for those variation points that are bound at build-time. In packaging the previous example system the distributor would either have to deny the JPG/PNG feature to the users, or force it on users that don't need it, or deploy binary packages for each desired set of feature selections.

Contribution This paper demonstrates how build management and package management can be elegantly integrated in a single formalism. We show a build tool called *Maak* that constructs systems from descriptions in a simple functional language. By providing the right modularity and caching constructs, the desired deployment characteristics can be obtained. In addition, the Maak language makes it easy to describe build variants.

Overview The remainder of this paper is structured as follows. We give a brief overview of the Maak language in section 2. We show how variant builds are implemented in section 3. Deployment methods are shown in section 4. The state of the implementation is addressed in section 5, and related work in section 6. We end with concluding remarks and directions for future work in section 7.

2 The Maak System

The task of a build manager is to generate *derivates* from *sources* by invoking the right *generators* (such as compilers), according to some formal specification of the system. The specification defines the actions by which derivates can be produced from sources or other derivates; that is, actions can depend on the outputs of other actions. The job of the build manager is to find a topological sort of the graph of actions that performs the actions in the right order.

There are several important aspects to such a tool. First, it should be *correct*: it should ensure *consistency* between the sources and the derivates, i.e., that all derivates are derived from current sources, or are equal to what they would be if they had been derived from current sources

Second, it is desirable to have a degree of *efficiency*: within the constraints of consistency redundant recompilations should be avoided. It is debatable whether this is the task of the build manager: it can be argued that this problem is properly solved in the actual generators, since only they have complete dependency information. Nevertheless, the efficiency aspect is the main purpose of classic build managers such as Make [1].

Third, the system specification formalism should enable us to easily specify *variants*. That is, we want to parameterise a (partial) specification so that the various members of a *software product line* (a set of systems sharing a common set of features) can be instantiated by providing different parameters.

2.1 Maak

Maak[1] is a build manager. It allows system models to be described in a simple functional language. Maak evaluates an expression that describes a *build graph*—a structure that describes *what* to build and *how* to build it—and then *realises* that graph by performing the actions contained in it.

Let's look at a real-life example. The ATerm library [3] is a library for the manipulation and storage of tree-like data structures. The library is fairly small, consisting of a dozen C source files. An interesting aspect of the package is that the library has to be built in several variants: the regular library, the library with debug information enabled, the library with *maximal subterm sharing* (a domain feature) disabled, and various others.

Support for this in the package's Makefile is rather *ad hoc*. To prevent the intermediate files used in building each variant from overwriting each other, the use of special filename extensions has to be arranged for:

```
SRCS = aterm.c list.c ...

libATerm_a_LIBADD     = $(SRCS:.c=.o)
libATerm_dbg_a_LIBADD = $(SRCS:.c=-dbg.o)
libATerm_ns_a_LIBADD  = $(SRCS:.c=-ns.o)
```

This fragment causes the extensions `.o`, `-dbg.o`, and `-ns.o` to be used for the object files used in building the aforementioned variants. To make it actually work, we have to provide pattern rules:

```
%.o: %.c
  gcc -c $< -o $@

%-dbg.o: %.c
  gcc -g -c $< -o $@

%-ns.o: %.c
  gcc -DNO_SHARING -c $< -o $@
```

Also note that this technique does not cover all variants in the variant space; for example, no library *without* maximal sharing but *with* debug information can be built.

The ATerm library can be built using Maak as follows. First, we consider the simple case of building just the regular variant:

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

atermLib = makeLibrary (srcs);
```

What happens here is that we define a list `srcs` of the C sources constituting the library. The variable `atermLib` is bound to the library that results from

[1] From the Dutch verb, "to make".

applying the function `makeLibrary` to the sources; `makeLibrary` is smart enough to compile the sources before putting them in the library.

It should be noted that `atermLib` is a variable name and not a filename, unlike, e.g., `./aterm.c`; the sole difference between the two syntactical classes is that filenames have slashes in them. So what's the filename of the library? The answer is that we don't need to know; we can unambiguously refer to it through the variable `atermLib`.

Hence, we can now *use* the library in building an executable program:

```
test = link {in = ./test.c, libs = atermLib};
```

We see here that Maak has two calling mechanisms: *positional parameters* (e.g., `f (x, y, z)`) and by passing an *attribute set* (`f {a = x, b = y, c = z}`). The latter allows arguments to be permuted or left undefined.

Similarly, we can create the debug and no-maximal-sharing variants:

```
atermLibDbg = makeLibrary {in = srcs, cflags = "-g"};
atermLibNS = makeLibrary {in = srcs, cflags = "-DNO_SHARING"};
```

To solve the variability issue comprehensively, we can abstract over these definitions by making a `atermLib` into a *function* with arguments `debug` and `sharing`:

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

after which we can select the desired variant:

```
test = link
  { in = ./test.c
  , libs = atermLib {debug = true, sharing = false}
  };
```

2.2 Language Overview

Maak's input formalism is a lazy functional language, meaning that variable bindings and function arguments are evaluated only when actually needed. This has two advantages. First, it prevents unused parts of the build graph from being evaluated. Second, it allows the definition of *control structures* in the language itself. For example, an if-then-else construct can take the form of a regular function `if` taking three arguments: the conditional and the values returned on true and false, respectively; only one of the latter is evaluated.

Types Maak has a number of basic data types:

- *Attribute sets* are sets of (name, value) pairs, i.e., records. The syntax for defining attribute sets is somewhat peculiar: attributes are in scope of the succeeding attributes. That is, in {`x = "a", y = x`} the attribute `y` has value `"a"`. Attributes can be selected by name, e.g., `attrs.x`.
- *Lists* are defined by juxtaposition between square brackets: `["x" "y" "z"]`.
- *Strings*. Strings are surrounded by single quotes, e.g., `'Hello'`. Double quotes are sugar for lists of strings. For example, `"gcc -c foo.c"` is sugar for `['gcc' '-c' 'foo.c']`. Additionally, arbitrary expressions can appear in list-of-string syntax by putting them between braces, allowing for easy argument list formation: `"gcc -o {out} -c {in}"` is sugar for `['gcc' '-o' out '-c' in]`.

Build Graphs Build graphs consist of two types of nodes: *file nodes* and *action nodes*. File nodes are represented as attribute sets. For example, {`type = "file", name = "/foo/bar"`}. Derivates have a attribute `partOf` whose value is the action that builds the derivate; that is, it specifies an edge in the graph.

The user can also write files using a more convenient syntax: e.g., `/foo/bar` is sugar for the previous example. Internally, Maak uses only absolute filenames. Relative names (e.g., `./foo.c`) are absolutised relative to the Maakfile in which they occur. The use of absolute names ensures that names are valid in all contexts. For example, the action `"gcc -I../include foo.c"` is fragile because it only works in the current directory. On the other hand, `"gcc -I{../include} {./foo.c}"` works everywhere.

Action nodes are also represented as attribute sets. There are several kinds of attributes involved in an action. File attributes denote either the action's sources or its derivates (i.e., they denote graph edges). A file x is a derivate of an action y if x`.partOf == y`, where `==` denotes pointer equality. The special attribute `build` specifies the command to be executed to perform the action.

We can therefore describe the action of generating a parser from a Yacc grammar as follows:

```
parser =
  { in = ./parser.y
  , csrc => ./parser.c
  , header => ./parser.h
  , build = "yacc -b {in}"
  # Output names are derived by Yacc from the input name.
  };
```

The notation `=>` is sugar: it ensures that the `partOf` attribute of the value points back at the defining attribute set, i.e., it defines a derivate of the action. Sources are not marked in a particular way. Given the action `parser`, the C source can be selected using the expression `parser.csrc`.

Defining Tools Rather than write each action in the graph explicitly, we can abstract over them, that is, we can write a function that takes a set of attributes and returns an action. The following example defines a basic C compiler function that takes two arguments: the source file `in`, and the compiler flags `cflags`.

```
compileC = {in, cflags}:
  { in = in
  , cflags = cflags
  , out => prefix (in) + '.o'
  , build = "cc -c {in} -o {out}"
  }.out;
```

The syntax {*args*}: *body* denotes a function taking the given arguments and returning the body. We select the `out` attribute of the action to make it easier to pass the output of one action as input into another (e.g., `link (compile (./foo.c))`).

Chaining For reasons of convenience and abstraction, we do not want to specify every intermediate step involved in building a target. For example, to build a program from C sources we would rather write `link ([./foo.c ./bar.c])` than `link ([(compileC (./foo.c)) (compileC (./bar.c))])`. The latter is bad because it exposes *how* we create an executable, namely, by per-module compilation. For example, if we would want to use a whole-program compiler instead, we would have to change the model, thus breaking abstraction.

The solution is *chaining*: automatically force arguments to be in such a format that they are acceptable (e.g., a linker doesn't accept C code but it does accept compiled C code). Note that Make supports chaining through *implicit rules*, but these work in the wrong direction. For example, `program: foo.o bar.o` specifies a program in terms of *object files* rather than source files.

Maak does not provide chaining as a language feature; rather, the tool definitions need to do it explicitly, in whatever way the author of the definition considers useful (i.e., Maak is *policy-free* in this regard).

For example, here is a `compileC` function that ensures that its argument is a C source file, by applying `yacc` or `flex` on the input if necessary.

```
compileC = forceC | {in, cflags}: ...

forceC = {in}: {args, in =
  try ([yacc flex (failIfNot (".c"))], in)}
```

Some language features need to be explained here. The `|` operator ("pipe") applies two functions in sequence, that is, `(f | g) (x) = g (f (x))`. The function `forceC` has the responsibility of converting the input into the appropriate format, but it should *fail* if that is not possible, to allow backtracking. This is what `try` does: it applies the specified functions to a value in order until one does not return the special value `fail`, and returns `fail` itself if none of them succeed. If the input is already a C file, we succeed trivially (in `failIfNot`).

Finally, the special variable `args` refers to the full set of arguments passed to a function, not just the formal arguments, while the construct {`args, in = ...`} returns an attribute set consisting of all original attributes except that `in` is replaced with the given value. This feature can be used for *attribute propagation*. For example, if a function `link` calls `compileC`, we do not want to declare `cflags` an argument to `link`, as that would break encapsulation. Using `args` we can generically pass down arbitrary arguments.

An apparent problem with this approach is that it won't automatically use functions that convert other file types into C; they have to be added to `forceC` (in contrast to Make, which uses implicit rules regardless of where they are defined). But on closer inspection, this is exactly right: the result of a function should depend only on its arguments, not on unrelated definitions in the code, since that makes it hard to predict the behaviour of functions. If extensibility is required, we can always pass in a convertor function as an argument to `compileC`.

Module System Maakfiles are modular: they can import other Maakfiles. For example,

```
import ./src/Maakfile;
```

makes the definitions in `./src/Maakfile` visible in the current Maakfile. The argument to the `import` keyword is an *expression* (rather than a filename) that evaluates to a filename. Crucially, as we shall see in section 4, this allows arbitrary module management policies to be defined by the user (rather than have them hard-coded in the language).

3 Variability

We saw in the previous section that Maak makes it easy to specify build variants. In this section we discuss how build variability is implemented in Maak.

A problem in building variants is that we have to prevent the derivates from each variant from overwriting each other; hence, they should not occupy the same names in the file system. There are several solutions to this problem. The approach taken by, e.g., Amake [4] is to build the derivates *in situ*, that is, in the location where the tools "natively" expect them; move them to a cache after they have been build; and move them back when they are needed as input to another action.

We take a somewhat simpler approach: the actions (i.e., the tool definition functions) are responsible for choosing output filenames such that variants do not overwrite each other. The usual approach is to form an output name using a *hash* of the input attributes. For example, we use the following definition for `compileC`:

```
compileC = {in, cflags}:
  { in = in
  , cflags = cflags
```

```
, out => anonFile (".o")
, build = exec ("{cc} -c {in} {cflags} -o {out}")
}.out;
```

The function `anonFile` computes a hash of the input attributes and uses that to form a filename. For example, `compileC (./foo.c)` might yield a filename `.maak_foo_305c.o`, while `compileC {in = ./foo.c, cflags = "-g"}` would yield `.maak_foo_54db.o`; in actuality, we use longer hashes to decrease the probability of a collision. But what happens when a collision occurs? In that case, a derivate may overwrite an older derivate, but since Maak registers the attributes used to build them, this will not lead to unsafe build; if the older derivate is required again, it will be rebuilt.

4 Deployment

As stated in the introduction, software deployment generally proceeds as follows. First, the system is *built* using, e.g., compilers, generally under the control of a build manager such as Make. Then, the relevant artifacts are *packaged*, that is, put in some deployable unit such as a zip-file or an RPM package; depending on the mechanism meta-data can be added to describe package dependencies and so on. Finally, the package is *installed*, typically by an *installer* shipped as part of the package, or by a *package manager* present on the target system such as the RedHat Package Manager.

Let's consider an example based on the ATerm library mentioned in section 2. The ATerm distribution consists of the library along with a set of utility programs. Suppose that we wish to split this distribution into two separate packages: `aterm` (containing the library), and `aterm-utils` (containing the utility programs). The typical source deployment process under Unix is:

1. Fetch the source code for package `aterm`.
2. Configure and build it (e.g., using Make).
3. Install it. This entails copying the library and C header files to some "global" location, such as `/usr/lib` and `/usr/include`.
4. Fetch the source code for package `aterm-utils`.
5. Configure and build it. Configuration includes finding or specifying the location of the `aterm` library; for example, Autoconf configuration scripts scan a number of well-known directories for library and header files.
6. Install it. This means copying the programs to, e.g., `/usr/bin`.

What is wrong with this approach? It is the way `aterm-utils` depends on `aterm`. It's not a formal dependency (i.e., it's not made explicit); rather, `aterm-utils` uses some artifacts (say, `libATerm.a`) that have (hopefully) been installed by `aterm`. But these are *uncontrolled* resources: how do we know that they are of the right version, built with the right parameters, etc.?

In this section we show a simpler approach to the deployment process. The central idea is to deploy packages in source form, i.e., along with an appropriate

Maakfile. Packages can depend on each other by having the Maakfile import the Maakfiles of other packages.

Figure 1 shows the Maakfile for the `aterm` package. It exports a function `atermLib` that builds a variant of the ATerm library, along with a pointer to the header files. Note that no installation occurs; the source is the installation.

```
import stdlibs; # for makeLibrary etc.

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

atermInclude = ./;
```

Fig. 1. Maakfile for package `aterm`

Figure 2 shows the Maakfile for the `aterm-utils` package. It imports the library package through the statement `import pkg ("aterm-1.6.7-2")`. The function `pkg` maps abstract package names to Maakfiles, while ensuring that the package is present on the system.

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

default = progs;

progs = [termsize ...];

termsize = link' (./termsize, ./termsize.c); # and so on...

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

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

Fig. 2. Maakfile for package `aterm-utils`

An Example Deployment Strategy Now, how do we deploy this? It's a matter of defining an appropriate implementation for the function `pkg`. It should be emphasised that `pkg` is not a primitive: it is a regular function. The user can define other functions, or change the definition of `pkg`, to obtain arbitrary package management policies. A possible implementation is outlined in figure 3. Here the source code is obtained from the network; in particular, it is checked out from a *Subversion* repository[2]. The source code for package X is downloaded into `/var/pkg/`X on the local machine, and `pkg` will return `/var/pkg/`X`/Maakfile` to the `import` statement.

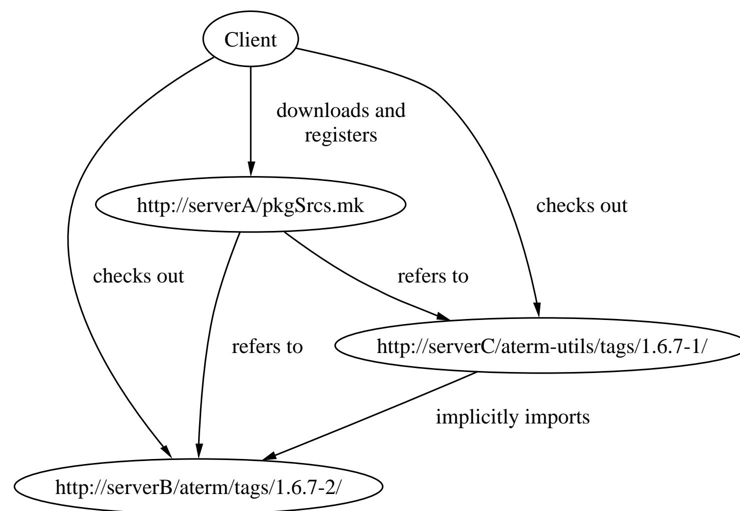

Fig. 3. A deployment strategy

Of course, `pkg` needs to know how to map package names to URLs. This mapping is maintained locally: through the function `registerPkg (`*`package-name`*`,` *`url`*`)` we can associate a package name with a URL. These mappings can also be obtained over the network by fetching a Maakfile containing calls to `registerPkg` from the network and executing it (clearly, security issues need to be addressed in the future!). The package `aterm-utils` can now be built by issuing the command

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

which will recursively obtain the source for `aterm-utils` and `aterm`, build the variant of `atermLib` required for the utilities, and finally build the utilities.

[2] Subversion is a version management system intended to be "a compelling replacement for CVS in the open source community" [5]. It fixes CVS's most obvious deficiencies, such as non-versioned directories and non-atomic commits.

(The switch `-f` obtains a Maakfile from the given expression rather than from the current directory).

Installation The above command will build the ATerm utilities, but it will not "install" them. In the Make paradigm, it is customary to have a *phony* `install` target that copies the appropriate files to the right system directories. This is essentially a redundant step. Indeed, it's just an additional complication (for example, it is often quite troublesome to get executables using dynamically linked libraries to work both in the source and installed location).

The main point of installing is to make software available to the user; for example, copying a program to `/usr/bin` has the effect of having it appear in every user's search path. That is, the point of installing is to *activate* the software. For example, the function `activateExec` will create a symbolic link in `/usr/bin` to its argument. Hence, the command

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

will build the utilities and make them available to the user. (The function `map` used in figure 2 applies a function—here, `activateExec`—to all elements of a list).

Binary distribution Of course, we cannot expect the clients to build from source, so we need the ability to transparently export derivates to the client; if a client runs Maak to build derivates that have already been built, i.e., were built with the same attributes, then the pre-existing ones will be used. On the other hand, if the client attempts to build a derivate with attributes or sources such that no equivalent derivate exists in any cache, it must be built locally. This enables a graceful fallback from *binary* distribution to *source* distribution.

Maak provides a primitive implementation of this idea. The command

```
maak 'exportDerivates (/tmp/shared, foo)'
```

will copy all derivates occurring in the build graph defined by the variable `foo` to the directory `/tmp/shared`, where a mapping is maintained from build attributes to files. Subsequently, another user can build `foo` through the command `maak --import /tmp/shared foo`; Maak will try to rebuild missing derivates first by looking them up in the mapping, and by rebuilding them if they do not occur in `/tmp/shared`. Therefore, if any changes have been made to the sources of `foo`, or to the build attributes, the derivates will be rebuilt.

5 Implementation

A prototype of Maak has been implemented and is available under the terms of the GNU Lesser General Public License at `http://www.cs.uu.nl/~eelco/maak/`. The implementation comes with a (currently small) standard library providing tool definitions for a number of languages and tools, include C and Java. The prototype is written in Haskell, a purely functional programming language.

This is a nice language for prototyping, but ultimately a re-implementation in C or C++ would be useful to improve portability and efficiency.

Maak implements up-to-date checking by tracing derivates per directory in a hidden file mapping filenames to the attributes used to build them, along with exact timestamps (and optionally, hashes of the contents) of input files.

A useful feature of the prototype is the ability to perform *build audits* on Linux systems to verify the completeness of Maakfile dependencies. By using the `strace` utility Maak can trace all `open()` system calls, determine all actual inputs and outputs of an action, and complain if there is a mismatch between the specified and actual sets of inputs and outputs.

Another useful feature are *generic operations* on the build graph: given a build graph, we can, for example, collect all leaf nodes to *automatically* create a source distribution, or collect all nodes that are *not* inputs to actions to create a binary distribution.

6 Related Work

Build Managers The most widely used build manager is Make [1], along with a large number of clones, not all of them source-compatible. Make's model is very simple: systems are described as a set of rules that specify a command through which a list of derivates can be created from a list of sources. Make rebuilds a derivate if any of the sources has a newer timestamp (a mechanism that is in itself subject to race conditions). Unfortunately, Make often causes inconsistent builds, since Makefiles tend to specify incomplete dependency information, and the up-to-date detection is unreliable; e.g., changes to compiler flags will not trigger recompilation. Make's input language is also quite simplistic, making it hard to specify variants.

The Makefile formalism is not sufficiently high-level; it does not provide scalable abstraction facilities. The abstraction mechanisms — variables and pattern rules — are all global. Hence, if we need to specify different ways of building targets, we cannot use them, unless we split the system into multiple Makefiles. This, however, creates the much greater problem of incomplete dependency graphs [6].

There have been attempts to fix this defect by building layers on top of Make rather than replace it, such as Automake [7], which generates Makefiles from a list of macro invocations. For example, the definition `foo_SOURCES = a.c b.y` will cause Automake to generate Make definitions that build the executable `foo` from the given C and Yacc source, install it, create a source distribution, and so on. The problem with such generation tools is that they do not shield the user from the lower layers; it is the user's job to map problems that occur in a Makefile back to the Automakefile from which it was generated. Automake does not provide a module system and so does not solve the problem of incomplete dependency graphs. It provides some basic variability mechanisms, such as the ability to build a library in several variants. However, Automake is not extensible, so this feature is somewhat *ad hoc*.

Autoconf [8] is often used in conjunction with Make and/or Automake to specialise an element of a product line automatically for the target platform. It is typically used to generate Makefiles from templates with values discovered during the configuration process substituted for variables. The heuristic approach to source configuration promoted by Autoconf is very useful in practice, but also unreliable. For example, we should not *guess* whether `/usr/lib/libfoo.so` is really the library we're looking for; rather, we should import the desired version of the library so that the process can never go wrong.

Autobundle [9] is a tool to simplify composition of separately deployable Autoconf-based packages. Based on descriptions of package dependencies, locations, etc., Autobundle generates a script that fetches the required source packages from the network, along with a configuration script and a Makefile for the composed package. This is similar to the package management strategy described in section 4, but it is yet another layer in the construction and deployment process.

A handful of systems go beyond Make's too-simple description language. Vesta [10] integrates version management and build management. It's *Software Description Language* is a functional language [11], similar to Maak's. An interesting aspect is the propagation of "global" settings (such as compiler flags), which happens by passing down an *environment* as a hidden argument to every function call; the environment is bound at top-level. In Maak propagation is explicit and left to the authors of tool definitions. Vesta also allows derivates to be shared among users; if a user attempts to build something that has previously been built by another user, the derivates can be obtained from a cache. This is quite reminiscent of our stated goal of allowing transparent binary deployment. However, the Vesta framework only allows building from immutable sources; that is, all sources to the build process (such as compilers) must be under version control. When deploying source systems, unfortunately, we cannot expect the recipient to have a identical environment to our own.

Odin [12] also has a somewhat functional flavour. For example, the expression `hello.c` denotes a source, while `hello.c :exe` denotes the executable obtained by compiling and linking `hello.c`; variants build can be expressed easily, e.g., `hello.c +debug :exe`. However, tool definitions in Odin are special entities, not functions.

Amake [4] is the build tool for the Amoeba distributed operating system. Like Odin, it separates the specification of build tools and system specifications. Given a set of sources Amake automatically completes the build graph, that is, it finds instances of tool definitions that build the desired targets from the given sources. This is contrary to the model of explicit tool application to values in Vesta and Maak. The obvious advantage is that specifications become shorter; the downside is that it becomes harder to specify alternative ways of building, and to see what's going on (generally, it is a good idea to be explicit in saying what you want).

The PROTEUS Configuration Language [13] aims to *model* all variability occurring in a system; Maak merely aims to implement it. Due to this smaller

scope Maak's input language is simpler: for example, PROTEUS has a **family** entity to describe product families; in Maak a product family is just a function from a set of parameters to the resulting artifacts.

Package Managers There are many package management systems, ranging from the basic— providing just simple installation and uninstallation facilities for individual packages—to the advanced—providing the features needed for ensuring a consistent system. The popular RedHat Package Manager (RPM) [2], used in several Linux distributions, is a reasonably solid system. By maintaining a database of all installed packages, it ensures that packages can be cleanly uninstalled, do not overwrite each others files, allows tracibility (e.g., to what package does file X belong?), verifies that the prerequisites for installation of a package (specified by the developer in an *RPM specfile*) are met, and so on.

But RPM also clearly demonstrates the dangers of separating build and package management: RPM packages often have incomplete dependency information [14]. For example, a package may use some library `libfoo.so` without actually declaring the `foo` package as a prerequisite. This cannot happen in Maak because the only way to access the library is by importing package `foo`.

Not a failing of RPM per se but of RPM package builders (and, indeed, most Unix packaging systems) is the difficulty of having several variants of the same product installed at the same time; e.g., RPMs of different versions or variants of Apache typically all want to be installed in `/usr/lib/apache/`. This is mostly a "cultural" problem: a better installation scheme (such as described in section 4) solves this problem. For example, a very useful feature for system administrators is the ability to query to what package a file belongs. This query becomes trivial if every package X is installed in `/var/pkg/`X.

There also exist several *source-based* package management systems, such as the *FreeBSD Ports Collection* [15]. The main attraction is that the system can be optimised towards the platform and requirements of the user, e.g., by selecting specific compiler optimisation flags for the user's processor, or by disabling unnecessary optional package features. The obvious downsides are slowness of installation, and that validation becomes hard: with so many possible variants, how can we be sure that the system compiles correctly, let alone runs correctly? For most ports pre-compiled *packages* exist (which do not offer build-time variability, of course). The two modes of installation are not abstracted over from the user's perspective; i.e., both present different user interfaces.

7 Conclusion

In this paper we have shown how build management and package management can be integrated. A prototype has been implemented. In the remainder of this section some issues for future work will be sketched.

The most important issue is that the sharing of derivates—essential for transparent binary/source distribution—is currently rather primitive. Any change to the attributes will invalidate a derivate. This is too rigid. For example, currently

recompilation will be triggered if the recipient has a different C compiler (since it's a dependency of the build process). A related issue is that we need to be able to do binary-only distributions. This could be done by making Maak pretend that the source does exist (e.g., by supplying file content hashes). Security issues related to derivate sharing need to be addressed as well. For example, if the administrator has built a package, other users should use it; but not the other way around.

Maak currently deals with build-time dependencies. Of course, we should be also able to handle run-time dependencies, such as the use of shared libraries.

Acknowledgments This work was supported in part by the Software Engineering Research Center (SERC). I am grateful to Eelco Visser, Andres Löh, and Dave Clarke for commenting on drafts of this paper.

References

1. Feldman, S.I.: Make — a program for maintaining computer programs. Software — Practice and Experience **9** (1979) 255–65
2. Bailey, E.C.: Maximum RPM. Sams (1997)
3. van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.: Efficient annotated terms. Software—Practice and Experience **30** (2000) 259–291
4. Baalbergen, E.H., Verstoep, K., Tanenbaum, A.S.: On the design of the Amoeba configuration manager. In: Proc. 2nd Int. Works. on Software Configuration Management. Volume 17 of ACM SIGSOFT Software Engineering Notes. (1989) 15–22
5. CollabNet: Subversion home page. `http://subversion.tigris.org` (2002)
6. Miller, P.: Recursive make considered harmful (1997)
7. Free Software Foundation: Automake home page. `http://www.gnu.org/software/automake/` (2002)
8. Free Software Foundation: Autoconf home page. `http://www.gnu.org/software/autoconf/` (2002)
9. de Jonge, M.: Source tree composition. In: Seventh International Conference on Software Reuse. Number 2319 in Lecture Notes in Computer Science, Springer-Verlag (2002)
10. Heydon, A., Levin, R., Mann, T., Yu, Y.: The Vesta approach to software configuration management. Technical Report Research Report 168, Compaq Systems Research Center (2001)
11. Heydon, A., Levin, R., Yu, Y.: Caching function calls using precise dependencies. In: ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, ACM Press (2000) 311–320
12. Clemm, G.M.: The Odin System — An Object Manager for Extensible Software Environments. PhD thesis, University of Colorado at Boulder (1986)
13. Tryggeseth, E., Gulla, B., Conradi, R.: Modelling systems with variability using the PROTEUS configuration language. In Estublier, J., ed.: Software Configuration Management: selected papers — ICSE SCM-4 and SCM-5 Workshops. Volume 1005 of Lecture Notes in Computer Science (LNCS)., Springer (1995)
14. Hart, J., D'Amelia, J.: An analysis of RPM validation drift. In: LISA '02: Sixteenth Systems Administration Conference, USENIX Association (2002) 155–166
15. The FreeBSD Project: FreeBSD Ports Collection. `http://www.freebsd.org/ports/` (2002)