Applying Module System Research to Package Management

David B. Tucker
Brown University
Box 1910
Computer Science Department
Providence, RI 02912
+1 401 863 7687
`dbtucker@cs.brown.edu`

Shriram Krishnamurthi
Brown University
Box 1910
Computer Science Department
Providence, RI 02912
+1 401 863 7722
`sk@cs.brown.edu`

ABSTRACT
Package managers found in popular operating systems are similar to module systems in many programming languages. Recent language research has focused on numerous ways to improve module systems, especially by endowing them with the characteristics of components. These improvements map well to package managers also. We identify several weaknesses with package managers, describe how components solve these problems in the programming context, and suggest how the structural principles of components can be applied to build the next generation of package managers.

Keywords
software package management, module systems, components, programming language modularity

1 INTRODUCTION

Software systems are growing increasingly complex and interdependent. For instance, numerous packages now offer alternate interfaces through mail clients, the Web, and other network devices, for both use and configuration. This increased functionality, however, translates directly into additional work to configure, install and maintain the software.

Managing software packages is thus a critical problem in real systems. Distributions of open-source operating systems, such as Debian, RedHat, and FreeBSD, all employ package management systems that aid the installation, configuration, and uninstallation of software. Currently, the main focus of such systems is to ensure that dependencies are correctly tracked; that is, if you want to install package A which relies on package B, then the package manager will require you to install B first.

Bundling existing software into packages and testing them is already a time-consuming and error-prone problem. This problem has intensified both with the proliferation of package systems, hardware platforms and OS kernels, and with the increasing complexity of the software packages themselves. The burden of tracking these dependencies seems to fall on the distribution maintainers, who must have an intimate knowledge of the individual packages. As a result, release dates are pushed back, or platforms are abandoned. Furthermore, *the main emphasis of OS distributions has become software packaging*, thus pushing aside other important goals such as usability and security.

One way to ameliorate these problems is to make it easier for programmers, distributors and users to specify, observe and satisfy complex dependencies. Existing package managers do a particularly poor job of meeting these ends. These tools tend to be ad hoc accretions of features. While these features reflect the diversity of packages and users, the tools themselves offer only limited support as *specification languages.* Without disregarding these features, it should be possible to provide package description and installation systems that better cater to the complexity of the underlying software.

In the same manner that software engineers have developed component-based solutions for organizing large software systems, they must now develop solutions for organizing large operating systems. We illuminate several shortcomings of real world package managers, and demonstrate how similar issues have been addressed in programming language research. We believe that the application of this research to software configuration will result in better package managers, and thus better systems overall. To focus our proposal, we will deal only with the installation process. Existing approaches to tasks such as upgrading, deletion and change tracking largely carry over in an analogous manner.

2 SHORTCOMINGS OF PACKAGE SYSTEMS

Current package managers are similar to simple module systems. They provide a simple encapsulation layer for a group of files, just as modules encapsulate a series of definitions. The requirement list of a package is similar to the import list of a module. Because there is no distinction between the package and its instances, a package does not possess any explicit state.

Packages are more complex than this simple characterization suggests. Each package does have a notion of state: typically, this is the configuration information stored on the filesystem. We consider this state because modifying it affects the behavior of the program even though the code itself remains the same.

> **Scenario #1:** You are the webmaster of a machine that runs two instances of `apache`, one as a production web server, and one as an experimental web server. Each copy runs on different ports, maintaining distinct logs.

If a package has state, it is meaningful to ask whether there is "really" only one copy of a package in a system. In fact, there can be several *instances* of a package, each with its own state, such as configuration information.[1] (In this paper, we will use Web servers as a running example.) Current package managers do not allow multiple copies of the same package. Furthermore, there is no straightforward way to extend current package managers to allow this capability; one fundamental assumption of these systems is that each package contains files that belong in exactly one place in the filesystem.

Many packages work around this on their own. For instance, production Web servers allow users to employ a variety of mechanisms, such as command-line flags, to specify multiple instances of the configuration of the server. These techniques are both unwieldy to the programmer and error-prone to the user: if the programmer fails to properly make the program "safe" for multiple instances, the user has to deal with the outcome. Early instances of the Apache server, for instance, allowed multiple log files for some services but not others. In short, these mechanisms are symptoms of the problem, not its solution.

A second problem that this simplistic notion of package obscures is that of mutual dependencies. Packages explicitly list a group of packages that they depend on, and the package manager checks for the existence of those packages before installing the new one. Sometimes, however, two packages may rely on the functionality of each other.

> **Scenario #2:** In Debian/GNU Linux, the packages `cron` and `exim` depend on one another. `cron` is a scheduler for periodic commands, the output of which are sent to the user by mail. `exim` is a mail server, which in turn relies on `cron` for scheduling periodic buffer flushes.

Traditional package managers cannot capture this mutual dependency. Expressing it directly usually results in installation failure: package A cannot install because it requires package B, but attempting to install B fails because it requires A. To work around this weakness, packages sometimes install dummy packages, much like forward declaration directives given to compilers. This solution is not only needlessly complex, it is also unsafe: there is no way to guarantee the placeholder will eventually be replaced. (Compilers, in contrast, can demand the entire program to verify satisfaction of the forward reference.) Once again, this is a symptom rather than a robust solution.

[1]This is the same distinction drawn between a *class* and its *objects*.

Third, packages are flat; they lack a hierarchical structure with client-specified satisfaction of dependencies.

> **Scenario #3:** As a systems administrator, you want to create partially complete distributions. You fix most of the dependencies of the Web server, but want to leave a few, such as ports and root filesystems, unspecified so that local deployers in the organization can define them.

Hierarchical structure makes it possible to create compound packages, which specify some but not all of a package's imports. Allowing clients to specify what satisfies an import makes it easy to reflect local concerns. In short, it moves control from the producer to the consumer.

3 COMPONENTS AND PACKAGE MANAGEMENT

Much of the research of the past several years at the intersection of software engineering and programming languages concerns the development of the notion of *components* [1, 2, 3, 4]. A careful examination of the scenarios listed above reveals an interesting detail: all these problems are addressed in programming languages by components. To wit, Szyperski, Flatt and others define components to have the following properties:

- they can be compiled separately;
- they are multiply instantiable;
- they are linked externally; and,
- their linkage is hierarchical.

In this paper, we exploit a specific implementation of components called *units* [1]. Units have been developed and used extensively in an extension of the Scheme programming language. Our extensive experience using units suggests that the unit model fits many of the needs of package management also. The rest of this paper elaborates on this position.

4 UNITS FOR PACKAGE MANAGEMENT

We first present units purely in the context of structuring the innards of programs. We then explain how these concepts address some of the problems raised in the context of package managers. Finally, we outline the steps we believe need to be taken to transport the unit model to package management.

A Primer on Units

The simplest way to understand a unit is as a closed container of code. Closure means that the code has no free variables, that is, all references to external entities are through explicit imports. This implies that the meaning of a unit is independent of its context of use, which in turn implies that it can be compiled separately.

A unit explicitly lists its imports and exports, which define its interface. Schematically, a unit looks like

```
(define Name
  (unit (import id ···)
        (export id ···)
      (define ...)
      ...))
```

That is, it imports a sequence of identifiers and exports another sequence of identifiers. The definitions can refer to one another, and to the imported identifiers. The exported identifiers must all have been defined within the unit. For instance, a Web server might have the form

```
(define Server
  (unit (import port initiate-logging log-entry)
        (export serve-directory)
      (define serve-directory ...)
      ...))
```

where *port* indicates which port to service and *initiate-logging* and *log-entry*, both produced by some log file maintenance application, are used by the server to log accesses. The server provides one service to clients: *serve-directory*. The client can invoke this function on any number of directory names. The server disseminates the content of each of these directories.

How does a programmer satisfy a unit's imports? The services imported by a unit must be exported by some other units. Therefore, all the programmer needs to do is to wire these producers and consumers together. To do this, a programmer writes a *compound unit*. In the running example, suppose the implementation of logging services is called *Logger*.

```
(define LoggingServer
  (compound-unit
    (import ws-port)
    (link
     [WS (Server ws-port (LOG start) (LOG do))]
     [LOG (Logger)])
    (export (WS serve-directory))))
```

This compound unit combines two units, the server and the logger. Each is given a tag, WS and LOG respectively, for the purpose of linking. The server has three imports. The first is left unresolved, making it an import of the compound unit. The other two are both satisfied by exports from the logger, denoted by the use of the logger's tag. The logger has chosen different names for these same services, so the logger's export of *start* is wired to the server's second import, *initiate-logging*; similarly, the logger's *do* is used to satisfy the server's third import, *log-entry*. The resulting compound unit exports the server's export.

To the outside world, compound units look similar to atomic units. Both have a sequence of imports and exports, while their contents are black-boxes. As a result, a compound unit may be used in any context expecting a unit; thus compound units may compound other compound units, as long as the imports and exports match up.

How do we use units? A programmer runs a unit by *invoking* it, supplying values for any remaining imports. Suppose we compound the logging server with a program that actually indicates which directories to serve:

```
(define Client
  (unit (import serve-dir)
        (export)
     (serve-dir "/etc/httpd/htroot")
     (serve-dir "/u/hckr/webstuff")))

(define ServeOurStuff
  (compound-unit
    (import ws-port)
    (link
     [LS (LoggingServer ws-port)]
     [CLIENT (Client (LS serve-directory))])
    (export)))
```

We can then run this program with

```
(invoke-unit ServeOurStuff 80)
```

which invokes a copy of the Web server on port 80 to serve the two directories named by the client.

In the scenarios of section 2, we discussed the problem of the two packages that have mutual dependencies. Our running example above has thus far ignored this complexity. The benefit of units, however, is that they can adequately handle mutual references, i.e., graph-shaped linking, just as well as they can handle conventional tree-shaped linking. Suppose the logger actually consumes the server's *serve-directory* function to indicate the location of the log files. The definition of *LoggingServer* then changes to read

```
(define LoggingServer
  (compound-unit
    (import ws-port)
    (link
     [WS (Server ws-port (LOG start) (LOG do))]
     [LOG (Logger (WS serve-directory) )])
    (export (WS serve-directory))))
```

(the boxed section shows the change from the previous version) and none of the other code needs to change. Thus, units not only accommodate mutual references, they do so without forcing changes to the interfaces of all the code that depends on the mutually referential entities.

This discussion of units obscures an important aspect of their dynamic semantics. How many copies of a unit are there in a running system? Each invocation of a (potentially compound) unit instantiates each of the constituent units. That is, the unit implementation creates a separate copy of the state of each unit on invocation. Programmers thus need not reason about shared state: a separate invocation is guaranteed to have a separate copy of the state, so they need not synchronize against other instances. This is again precisely the same as the class-versus-object distinction.

The description of the unit system has neglected pragmatic concerns such as name management. For instance, suppose the client links two servers with the same logging agent. Each server might export a procedure named *serve*. In the C linking context, this is problematic because clients do not have explicit control over the names. In contrast, in a compound unit, the two *serve*s can be distinguished by using the **link** clause tag names to tell them apart. This not only permits the linking of two servers, it also makes it easy for clients to indicate precisely which server to use in which context.

Other module languages do not have the same capabilities as units. In Java, for example, modules ("packages") are internally linked—the connection between modules is specified by the name in the `import` statement. ML's functors have external linking, but they do not support mutually recursive modules. C's module system (i.e., the system linker) appears to have both these capabilities. Linkage in C is external: the actual program pieces defined by these names can be interchanged via the linker. These modules can also refer to one another. The problem with name-based linking, though, is that programmers cannot instantiate a module multiple times; rather, each module's exports reside in a global space from which other modules atmospherically resolve their dependencies. Thus, C's implicit module system, while in the right spirit for extensible development, fails to provide sufficient control to programmers, and does not allow multiple uses of the same module in a linking specification. The unit system addresses all these limitations.

Addressing Deficiencies of Package Managers

Now we can examine how units can address each of the scenarios described in section 2.

1. The first scenario discusses the need to have multiple copies of the same package on a system. By now, the reader should recognize this popular refrain as the class-object distinction. Traditional package managers do not make this distinction in the same way that traditional program organization module systems have also failed to distinguish between the two. In a component framework, such as that implemented by units, this distinction is easy to make. It is precisely the difference between the units themselves, which are static entities, and their instances, of which there can be many dynamic ones.

2. The second scenario demonstrates how packages can refer to one another in their configurations. It should be clear that the unit-style linking mechanism permits mutual references at the unit level. The act of resolving references essentially amounts to a fixed-point computation; units provide a natural way of expressing the constraints, and resolve the references in their implementation.

3. The third scenario refers to creating partially satisfied groups of packages. The reader will recognize this as analogous to the compounding of the Web server with a particular implementation of logging. This fixes some parameters, namely those of the server that pertain to logging. It leaves some other parameters (in the example, the server's port) to be fixed by the actual client. Similar configurations help administrators make organization-wide policy decisions without having to commit all decisions at the same level.

5 IMPLEMENTING PACKAGES AS UNITS

In this section, we describe how each of the elements of units maps to package management.

Naturally, we identify atomic units with individual packages. A package has a set of requirements and provides itself as a service, which maps naturally to the imports and exports of units. Whereas in a programming context a unit's imports are functions and values, in the package context the natural analog seems to be other packages—in short, a dependency list. We consider the problem of versioning to be orthogonal to the problems we are trying to address here. Therefore, which packages are allowed to satisfy a unit's import is a function of the versioning semantics in use.

Normally, the body of an atomic unit contains definitions. The body of a package will probably instead contain the source and binary files, scripts, registry updates, and other information that constitutes the package. That is, a unit becomes the atomic notion of distribution; this would have to be a special file format, just as, say, Debian packages or RedHat RPMs are.

A compound unit is also a distributable bundle that should be indistinguishable from an atomic unit to a client. Therefore, the distribution form of a compound unit needs to contain copies of all the units (atomic or compound) that are linked together by it. An actual implementation should have the liberty of replacing the actual atomic units with references to them—for instance, when linking a logger and a Web server, the compound unit does not need to contain the entire server package, but can rather rely on the local copy available on the target host.

The important operation is unit invocation, since this truly distinguishes units from existing package managers. Invocation needs to create a fresh "heap" for the units to populate with their state. In this case, the state refers principally to files of various sorts. Therefore, the implementation of units needs to create a distinguished area in the disk system to hold the state of each invocation. For instance, on Unix, the unit-based package manager can assume control of a directory like `/etc/pkg/inst/`, and each invocation creates a sub-directory there with a distinguished name (say an instance number). Thus, the first invocation of a unit creates the directory `/etc/pkg/inst/1/`, and so forth. This then becomes the root directory for all files specific to that instance. If the Web server wants to create the file `/etc/httpd/access_log`, say, this gets written to `/etc/pkg/inst/1/etc/httpd/access_log`. Therefore, the state of one invocation remains completely separate from the state of another.

This raises the question of how to detect files corresponding to state and how to relocate them. Determining which files to relocate seems to be a package-specific task. In general, each package has some region where it stores files, and all file accesses in this region need to be relocated. In contract, references to files outside this region, or to known files such as `/etc/passwd`, should be left unmolested. Having determined which files to relocate, the actual modification can employ both static and dynamic components. Many of the paths can be modified statically by simply altering the pathnames. However, a package might also synthesize filenames at run-time. The common point of control for all these accesses is the *fopen* primitive; the implementation of the unit-based package manager needs to trap these invocations and, when they refer to local files, refer them elsewhere. The simplest way of accomplishing this is by altering the shared library that provides *fopen*. For some applications, it may be necessary to (also) modify the binary. In these cases, the modified, instance-specific binary can reside in the instance directory. While this relocation does impose a run-time overhead, we believe the overhead is negligible compared to the cost of actual file i/o operations.[2]

So far, we have only discussed configuration files as elements of state. In practice, an application has many other bits of state as well. For instance, it may listen on TCP ports; the identity of these ports is part of its state. Sometimes these ports correspond to well-known services (the default Web server should listen on port 80, for instance). In other cases, the instance simply listens on *some* port whose identity needs to be communicated with the user. These state resources can be re-routed just like files, and the interposed library layer can communicate the mapping from specified to selected ports to the user.

This re-routing of state has one very practical benefit. Currently, package writers and installers have to go to great pains to determine where to situate configuration, cache and other files. Each operating system, and even different distributions of the same operating system (such as Linux), expect applications files to reside in different locations. The problem is compounded by the expectations of individual sites, which may complicate or even override these conventions. Similar problems arise for resources such as port numbers.

A program distributed for the unit-based package manager can avoid these problems entirely. It can simply place configuration information in a (virtual) root directory, i.e., in `/` (under Unix). Why? Because the package system is going to relocate this file to a harmless directory such as `/etc/pkg/inst/42/` (since for most applications, writing to `/` will obviously need to be relocated). Further, two applications linked together can easily share files through this simple protocol: suppose the Web server and logging service need to share a configuration file, they only need to know its name, and can assume its location is `/`—because the package manager will map each program's `/` to the same directory. This avoids the complex and frustrating process of supplying compile-time or command-line flags, or setting environment variables, to share this information.

In short, units refine the notion of time in the system. Whereas previously packages had a static existence independent of program execution, units interpose the time of invocation. There are therefore both packages, which are entirely static entities, and invocations, which are the states corresponding to configurations of the packages. A running program executes the code from a package (or a copy thereof, customized to an instance), relative to the *instance*'s state.

6 CONCLUSION

We have discussed several failings of current package management systems. Elevating the packages to components can address these problems, which we illustrate with several examples. We have also briefly discussed how to implement an improved alternative for package management.

ACKNOWLEDGMENT

We thank Manos Renieris for numerous useful conversations and suggestions on the writing.

REFERENCES

[1] M. Flatt and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[2] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, March 1999.

[3] Y. Smaragdakis and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.

[4] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[2]This relocation seems like a task for Unix's *chroot* call, but *chroot* is too constraining; we want to relocate only some, not all, files.