Component Deployment with PLaneT

You Want it *Where?*

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Abstract

For the past two years we have been developing PLaneT, a package manager built into PLT Scheme's module system that simplifies program development by doing away with the distinction between installed and uninstalled packages. In this paper we explain how PLaneT works and the rationales behind our major design choices, focusing particularly on our decision to integrate PLaneT into PLT Scheme and the consequences that decision had for PLaneT's design. We also report our experience as PLaneT users and developers and describe what have emerged as PLaneT's biggest advantages and drawbacks.

1. Introduction

No matter how great your programming language is, it is always harder to write a program in it yourself than it is to let someone else write it for you. That is one reason why libraries are a big deal in programming languages, big enough that some languages are associated as much with their libraries as they are with their core features (*e.g.*, Java with `java.util` and C++ with the STL).

But when you move away from the standard libraries that come built in to your programming language, you can end up paying a high price for your convenience. If your program depends on any libraries, you have to make sure those dependences are installed wherever your program will run. And any dependences those libraries have need to be installed too, and any dependences *those* libraries have, and so on. If you don't have any help with this task, it is often easier to avoid using too many libraries than it is to explain how to install dozens of dependences in a `README` file. A Scheme programmer in this situation might reasonably bemoan the fact that the choice needs to be made: why, in a language like Scheme with so many great facilities for code reuse, is it so impractical to actually reuse code?

In this paper we describe our attempt at addressing that problem by presenting the PLaneT package distribution system built into PLT Scheme [5], a component-deployment tool that aims to make package distribution and installation entirely transparent. To the extent possible, PLaneT does away with the notion of an uninstalled package: a developer can use any PLaneT package in a program just by referring to it as though it were already installed, and whenever anyone runs that program, the PLaneT system will automatically install it if necessary. Effectively, programmers get to treat *every* library, even those they write themselves, as a standard library, giving them the advantages of libraries without the pain of library deployment.

Since PLaneT retrieves packages from a centralized repository, it also establishes a component marketplace for developers; it serves as a one-stop shop where they can publish their own programs and discover useful programs written by others. In effect, PLaneT is an infrastructure that transforms PLT Scheme's effective standard library from a monolithic cathedral development effort into a bazaar where anyone can contribute and the most useful libraries rise to the top, buoyed by their quality and usefulness rather than an official blessing [11]. This bazaar-like development reinforces itself to the benefit of all PLT Scheme users: in the two years since PLaneT has been included in PLT Scheme, users have contributed nearly 300,000 lines of new Scheme code in 115 packages.

In this paper, we explain our experience with building and maintaining PLaneT, with particular attention to PLaneT's more unusual design features and how they have worked in practice. Section 2 sets the stage by briefly introducing some existing component systems such as the Comprehensive Perl Archive Network (CPAN) with particular attention to their deployment strategies. Section 3 then uses those other systems to motivate our goals for PLaneT. The next few sections explain how we tried to achieve those goals: section 4 explains our server design, section 5 explains our client design with particular attention to our choice of integrating component deployment into the language rather than keeping it as an external program, section 6 discusses how PLaneT maintains PLT Scheme tool support, and section 7 explains a few complications we foresaw and tried to address in our design. In section 8 we try to assess how well our design has stood up over the two years PLaneT has been in use.

2. Some popular component systems

Before we explain the decisions we made for PLaneT, we need to explain some background on existing component systems, how they work, and some of their pitfalls. The software development community has learned a lot about both the benefits of software components and the problems they can cause. Probably the most famous such problem is so-called "DLL hell," a term coined to describe what can happen to a Microsoft Windows installation if DLLs (for "dynamically-linked libraries," a component technology) are installed incorrectly. Under the DLL system, when a program wants some functionality from a DLL, it refers to the DLL by name only. This can become a problem because the DLL may evolve over the course of new releases; if two different programs both rely on different versions of the same library, then they cannot both be installed on the same system at the same time. Furthermore, since DLLs have no particular package deployment strategy, soft-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

ware installers have to install any necessary libraries themselves; and if an installer overwrites a DLL it can inadvertently break other programs, possibly even leading to a situation where the user's entire system is unusable. Microsoft has addressed this problem in several ways; the most recent of which is its .NET assembly format, which includes metadata that includes versioning information [10].

Neither of those systems deal with component distribution — it is the programmer's responsibility to figure out some other way to locate useful components and to arrange for them to be installed on users' computers, and if a user wants to upgrade a component then it is up to that user to find and install it. Other component systems have done work to help with these tasks, though. For instance, CPAN [1] is a central repository of Perl libraries and scripts that users can download, and tools allow users to install and upgrade these libraries automatically in some cases (more on this later). The Ruby programming language features a similar system called RubyGems [12]. Many Unix-like operating systems also feature package distribution systems that address these problems; Debian's `dpkg` system [14], for instance, provides a central repository of programs and shared libraries that users can automatically fetch, install, and upgrade upon request.

Since these systems try to do more than DLLs do, they have to solve more potential problems. The most major of these is that if a tool installs a new component that relies on functionality from other components, the tool must also install appropriate versions of those prerequisites if the user does not already have them. Identifying the entire set of uninstalled prerequisites is not too difficult given the right metadata, but automatically deciding what "appropriate" means in this context is more of a challenge. For a human, the most appropriate version of a prerequisite would probably be the one that provides all the functionality that the requiring component needs in the highest-quality way and has the fewest bugs. But an automatic tool cannot hope to figure out which package that is, so it needs to simplify the problem somehow.

The solution `CPAN.pm` (an automated client for CPAN) uses is to assume that the version of a given package with the highest version number is the most appropriate, based on the reasoning that the highest version number represents the most recent package and therefore the one with the most features and fewest bugs. That is, if the user requests package `foo`, then `CPAN.pm` finds the most recent version of `foo` it can. Furthermore, if `foo` depends on package `bar`, then `CPAN.pm` downloads the highest-numbered version of `bar` available (unless `foo` explicitly asks for a particular version or a version number in a particular numeric range).

This policy is extremely optimistic, in that it assumes all programs can use any version of a package in place of its predecessors. If for instance `bar` removes a function from its public interface in a particular release, then unless `foo` compensates for that change by releasing a new package with updated code or dependence information, it will fail to install properly. In practice this problem does not come up much, probably because most packages on CPAN only release a few different versions and those that are intended for use as libraries try to maintain backwards-compatibility. However, it can and has come up in the past, and there is no automatic way to cope with it.

Debian's `dpkg` system uses a similar underlying strategy to `CPAN.pm`'s, but has evolved a convention that serves as a coping mechanism: many Debian packages include a number directly in their names (for instance, `libc5` versus `libc6`); if a package changes and breaks backwards-compatibility, the number in the package's name changes. This way, humans looking through the package list can select the most recent version of a package for new projects without worrying that future revisions will break backwards compatibility.

The RubyGems system takes the convention a step farther; their "Rational Versioning Policy," while not technically required of packages, is strongly recommended and explicitly supported by their automatic installation tools. The rational versioning policy requires that a package's version should be a dotted triple of numbers (*e.g.*, 1.4.2). Incrementing the first number indicates a backwards-incompatible change, incrementing the second number indicates a backwards-compatible change that adds features, and an increment of the last number indicates an internal change such as a bug-fix that should not be visible to users. The automatic installation tools use these numbers to decide which version of a package to download; if a package requests a package of version number at least 2.3.1, then the tool considers version 2.5.0 an acceptable substitute but not version 3.4.2.

All of these systems are in some sense optimistic, because they all let a tool decide to substitute one version of a package for another, and there is no way to know for certain that the program making the request doesn't depend on some hidden behavior that differs between the two. Still, in practice this system seems to work out, since most programs are not so deeply tied to the inner workings of libraries they depend on that changes to those libraries will break them.

3. Goals

In 2003 we decided to build a "CPAN for PLT Scheme" called PLaneT[1]. We wanted our design to keep or improve on the good features of existing component systems while removing as many of the undesirable properties of those approaches as we could. Specifically, we wanted to make it very easy for PLaneT to automatically retrieve and install packages and recursively satisfy dependencies with the best available packages, while giving package developers as much flexibility as possible. We also wanted to make it easy for programmers to find available packages and incorporate them into their own programs, and easy for users of those programs to install the packages they needed. More specifically:

- We wanted programmers to be able to find available libraries easily and should be able to correctly incorporate them into programs without having to know much about PLaneT. We also wanted PLaneT's more advanced features to have a gentle learning curve.
- We wanted users to be able to correctly use programs that rely on PLaneT libraries without being aware that those libraries, or PLaneT itself, existed. Ideally, we wanted whether or not a program relies on a library to be an implementation detail.

Moreover, and perhaps most importantly, we wanted to get rid of the *statefulness* inherent to other package management systems. With CPAN, for instance, every available package might or might not be installed on any particular user's computer, so a program that relies on a particular CPAN package might or might not work for that user depending on the global state defined by which packages are installed. If that state does not have the necessary packages, then the user or a setup script has to do what amounts to a **set!** that updates the state. Just as global variables make it hard to reason about whether a particular code snippet will work, we hypothesized that the statefulness of package installations make it more difficult than necessary to use component deployment systems. We wanted to eliminate that problem to the extent possible.

One non-goal we had for PLaneT was making the client able to manage the user's overall computing environment, such as man-

[1] After considering the name "CSPAN" briefly, we decided on the name PLaneT due to the fact that it implied a global network, it contained the letters P, L, and T in the correct order, and it turned out that John Clements had coincidentally already designed a logo fit the name perfectly.

aging global system configuration files or installing shared C libraries into standard locations. PLaneT is intended to help PLT Scheme programmers share PLT Scheme libraries effectively, and we can accomplish that goal much more simply if we assume that the PLaneT client can have absolute control over its domain. That isn't to say that PLaneT code cannot interact with C libraries — in fact, thanks to Barzilay's foreign interface [3] it is often quite easy to write a PLaneT package that interacts with a C library — but distributing and installing those C libraries on all the platforms PLT Scheme supports is not a problem PLaneT is intended to solve.

Our design for PLaneT consists of a client and a server: the client satisfies programs' requests for packages by asking for a suitable package file from the server, and the server determines the best package to serve for each request.

4. The PLaneT server

Most of the rest of this paper concerns the design of the PLaneT client, but there is something to be said about the server as well. PLaneT relies on a single, centralized repository located at `http://planet.plt-scheme.org/`. That site is the ultimate source for all PLaneT packages: it contains the master list of all packages that are available, and it is also responsible for answering clients' package requests. Naturally, that means it has to decide which package to return in response to these requests. Furthermore, since the server is centralized, it is responsible for deciding what packages can be published at all.

4.1 Versioning policy

As we discussed in section 2, PLaneT, or any other automatic component deployment system, needs a policy to make tractable the decision of what packages are compatible with what other packages. The policy we have chosen is essentially a stricter variant of Debian's de facto versioning policy or RubyGems' Rational Versioning Policy, with the difference that PLaneT distinguishes between the *underlying* version of a library, which the author may choose arbitrarily, and the *package* version, which PLaneT assigns. Because of this split, we can control the policy for package version numbers without demanding that package authors conform to our numbering policies for the "real" version numbers of their packages.

The package version of any package consists of two integers: the *major* version number and the *minor* version number (which we will abbreviate *major.minor* in descriptions, though technically version numbers 1.1, 1.10, and 1.100 are all distinct). The first version of a package is always 1.0, the next backwards compatible version is always 1.1, and then 1.2, and on up, incrementing the minor version number but keeping the major version number constant. If a new version breaks compatibility, it gets the next major version and gets minor version 0, so the first version that breaks backwards compatibility with the original package is always 2.0. The pattern is absolute: compatible upgrades increment the minor version only, and incompatible upgrades increment the major version and reset the minor version.

This makes it very easy for the PLaneT server to identify the best package to send to a client in response to any request. For instance, as of this writing the PLaneT repository contains four versions of the galore.plt package with package versions 1.0, 2.0, 3.0, 3.1, 3.2, and 3.3. If a client requests a package compatible with galore.plt version 3.0, the server can easily determine that 3.3 is the right package version to use for that request, based only on the version numbers. Similarly if a client requests a package compatible with version 2.0, then it knows to respond with version 2.0 even though more recent versions of the package are available, since the developer has indicated that those versions are not backwards-compatible.

The policy's effectiveness depends on the conjecture that package maintainers' notions of backwards compatibility correspond to actual backwards compatibility in users' programs. While it is easy to come up with hypothetical scenarios in which the conjecture would be false, it seems to hold nearly always in practice, and the fact that is a more conservative version of strategies already used in successful component deployment systems gives us more assurance that our policy represents a reasonable trade-off.

4.2 Package quality control

As maintainers of the PLaneT repository, we are responsible for deciding which submitted packages to include into the repository. We decided early on that our policy for quality control should be to accept all submitted packages. We decided this for a few reasons. First, we wanted to make it as easy as possible for authors to submit packages, because after all a package repository is only useful if it contains packages. All the nontrivial quality-control metrics we could think of would either entail too much work to scale effectively orimpose barriers to submission that we thought were unacceptably high. For instance, we considered only accepting packages that provided a test suite whose tests all passed, but we decided that would discourage programmers from submitting packages without first writing large test suites in some test-suite notation that we would have to devise; this seemed too discouraging. (Of course we *want* programmers to write large, high-quality test suites for their packages, and many of them do; but *mandating* those test suites seemed overly burdensome.) Second, we suspected that low-quality packages didn't need any special weeding out, since no one would want to use them or suggest that others use them; meanwhile high-quality packages would naturally float to the top without any help.

As for malicious packages, after a similar thought process we decided that there was nothing technical we could reasonably do that would stop a determined programmers from publishing a package that intentionally did harm, and that throwing up technical hurdles would likely do more harm than good by offering users a false sense of security and malicious programmers a challenge to try and beat. But again, we suspected that the community of package users could probably address this better by reputation: bad packages and their authors would be warned against and good packages and their authors would be promoted.

In short, after some consideration we decided that trying to perform any kind of serious quality control on incoming packages amounted to an attempt at a technical solution for a social problem, so we opted to let social forces solve it instead. This solution is the same solution used by the other component systems we studied, which gave us confidence that our decision was workable.

5. The client as a language feature

The PLaneT client works by hooking in to the guts of PLT Scheme's module system. In PLT Scheme, programmers can write modules that depend on other modules in several ways. For instance, the following code:

```
(module circle-lib mzscheme
  (require (file "database.ss")) ; to use run-sql-query
  (require (lib "math.ss")) ; to use π

  (define query "SELECT radius FROM circles")
  (define areas
    (map (lambda (r) (* pi r r)) (run-sql-query q)))

  (provide areas))
```

defines a module named *circle-lib* in the default *mzscheme* language. The first thing mzscheme does when loading *circle-lib* (either in response to a direct request from the user or because some other module has required it) is to process all of its requirements. In this case there are two, (**require** (**file** "database.ss")) and (**require** (**lib** "math.ss")), each of which use a different variant of the **require** form: the (**require** (**file** "database.ss")) variant loads the module in the file database.ss in the same directory as the source code of *circle-lib*, and (**require** (**lib** "math.ss")) loads the module in the file math.ss located in PLT Scheme's standard library.

PLaneT integrates into this system by simply adding another new require form. It looks like this:

```
(require (planet "htmlprag.ss" ("neil" "htmlprag.plt" 1 3)))
```

This declaration tells mzscheme that the module that contains it depends on the module found in file htmlprag.ss from the package published by PLaneT user neil called htmlprag.plt, and that only packages with major version 1 and minor version at least 3 are acceptable.

When mzscheme processes this **require** statement, it asks the PLaneT client for the path to a matching package, in which it will look for the file the user wanted (htmlprag.ss in this case). The client keeps installed packages in its own private cache, with each package version installed in its own separate subdirectory. It consults this cache in response to these requests; due to the PLaneT package version numbering strategy, it does not need to go over the network to determine if it already has an installed a package that satisfies the request. If it already has a suitable package installed, it just returns a path do that installation; if it does not, it contacts the central PLaneT server, puts in its request, and installs the package the server returns; the process of installing that package recursively fetches and installs any other packages that might also be necessary.

This strategy goes a long way towards solving the statefulness problem we explained in section 3, for the simple reason that we can think of the statefulness problem as a simpler variant of the problem modules were designed to solve. In the absence of modules (or an equivalent to modules), programmers have to write what amount to control programs that tell the underlying Scheme system how to load definitions in such a way that the top level is set to an appropriate state before running the main program — a problem that is especially bad for compilation [6], but that causes annoyance even without considering compilation at all. The module system addresses that top-level statefulness problem by making modules explicitly state their requirements rather than relying on context, and making it mzscheme's responsibility to figure out how to satisfy those requirements; from this perspective PLaneT just extends the solution to that statefulness problem to address the package-installation statefulness problem as well[2].

The connection between packages and the **require** statement also has a few pleasant side effects. First, it makes it easy for programmers to understand: PLT Scheme programmers need to learn how to use the module system anyway, so using PLaneT just means learning one new twist on a familiar concept, not having to learn how to use an entirely new feature. Second, including package dependence information directly in the source code means that there is no need for a PLaneT programmer to write a separate metadata file indicating which packages a program relies on. PLaneT is the only component deployment system we are aware of with this property.

[2] There is one important exception where statefulness shines though: if a computer that is not connected to the network runs a program that requires a PLaneT package, then that program might or might not succeed depending on whether or not the PLaneT client has already installed a suitable version of the required package. If it has, then the requirement will succeed; otherwise it will fail and signal an error. Of course there isn't very much the PLaneT client can do to prevent this stateful behavior, but it does mean that PLaneT works best for computers that are usually online.

6. Tool support

Since the PLaneT client is a part of the language of PLT Scheme, tools that work with PLT Scheme code need to be able to work with PLaneT packages just as naturally as they work with any other construct in the language. This is particularly important for DrScheme, the PLT Scheme development environment [4], because it provides programmers with several development tools that PLT Scheme programmers expect to work consistently regardless of which features their programs use. For instance, it provides a syntax checker that verifies proper syntax and draws arrows between bound variables and their binding occurrences, a syntactic coverage checker that verifies that a program has actually executed every code path, and several debuggers and error tracers. Making PLaneT a language extension rather than an add-on tool places an extra burden on us here, since it means programmers will expect all the language tools to work seamlessly with PLaneT requires. For this reason, we have tried to make tool support for PLaneT as automatic as possible.

In doing so, we are helped by the fact that PLaneT is incorporated into Scheme rather than some other language, because due to macros Scheme tools cannot make very strong assumptions about the source code they will process. Similarly, in PLT Scheme tools must be generic in processing **require** statements because a macro-like facility exists for them as well: even in fully-expanded code, a program may not assume that a **require** statement of the form (**require** *expr*) has any particular semantics, because *expr* has not itself been "expanded" into a special value representing a particular module instantiation. To get that value, the tool must pass *expr* to a special function called the module name resolver, which is the only function that is entitled to say how a particular require form maps to a target module. PLaneT is nothing but an extension to the module name resolver that downloads and installs packages in the process of computing this mapping; since tools that care about the meaning of **require** statements have to go through the module name resolver anyway, they automatically inherit PLaneT's behavior.

This has made DrScheme's tools easy to adapt to PLaneT, and in fact almost none of them required any modification. Figure 1 shows some of DrScheme's tools in action on a small PLaneT program: the first screenshot shows the syntax check tool correctly identifying bindings that originate in a PLaneT package, and the second shows PLaneT working with DrScheme's error-tracing and debugging tools simultaneously. None of these tools needed to be altered at all to work correctly with PLaneT.

We did have to make some changes to a few tools for a couple of reasons. The first was that some tools were only designed to work on particular require forms, or made assumptions about the way that libraries would be laid out that turned out to be too strong. For instance, both the Help Desk and the compilation manager assumed that all programs were installed somewhere within the paths that comprise PLT Scheme's standard library; PLaneT installs downloaded code in other locations, which caused those tools to fail until we fixed them. In principle, this category of tool change was just fixing bugs that had been in these tools all along, though only exposed by PLaneT's new use patterns.

The second reason we had to modify some tools was that the generic action that they took for all libraries didn't really make sense in the context of PLaneT packages, so we had to add special cases. For instance, we had to add code to the Setup PLT installation management tool so that it would treat multiple installed versions of the same package specially. Also, DrScheme also includes a module browser that shows a program's requirements graph; we modified that tool to allow users to hide PLaneT requirements in that display as a special case. The module browser worked with-

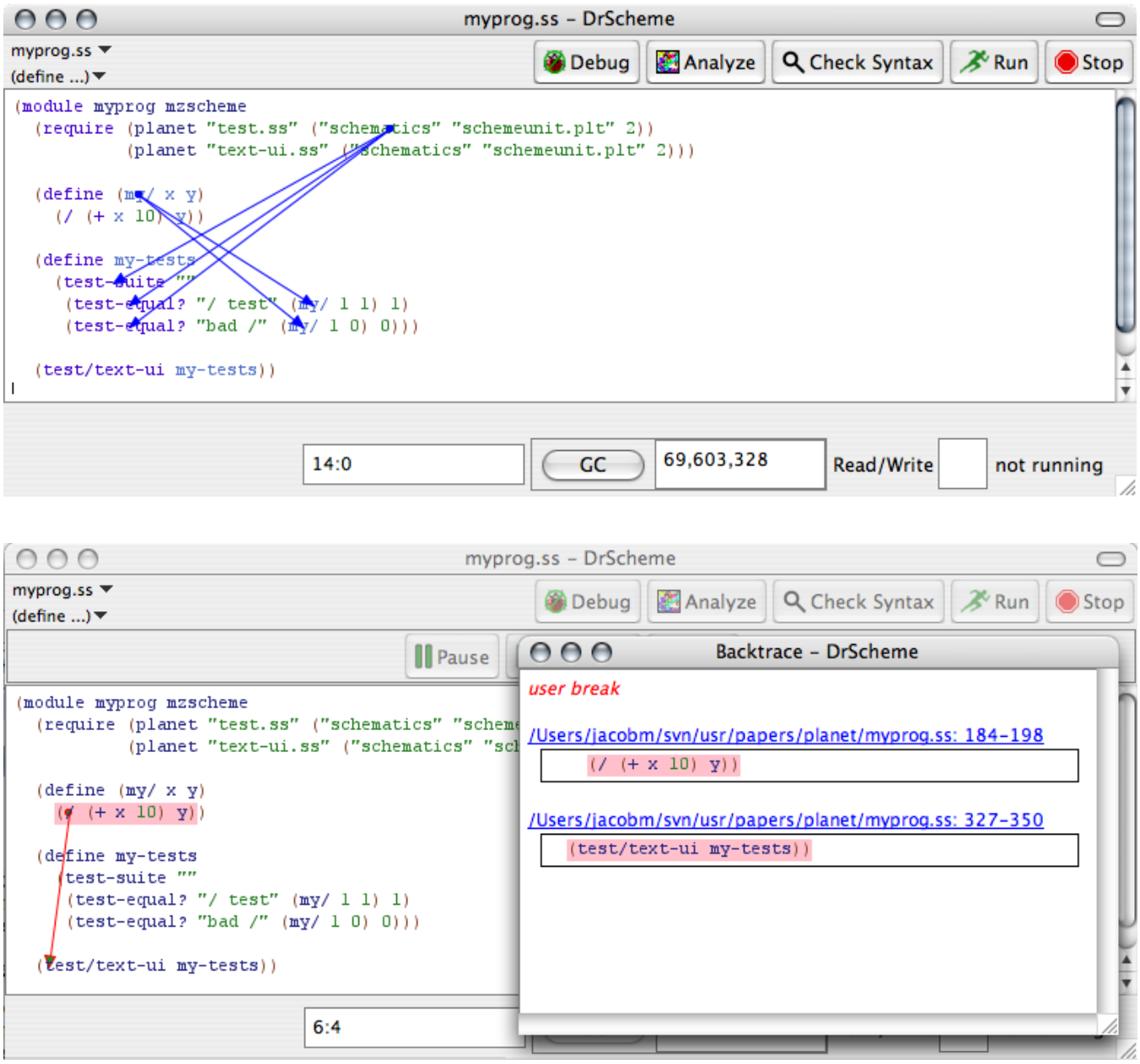

Figure 1. Two tools interacting seamlessly with PLaneT packages

out that modification, but we found that the extra requirements that PLaneT packages introduced transitively tended to add a lot of noise without being very useful. Figure 2 illustrates the impact this special case had on the quality of the module browser's output.

7. A few complications

We think PLaneT's version-naming policy works fairly well: it is simple to implement, simple to understand, and easy enough to use that if you don't understand it at all you'll probably do the right thing anyway. But of course versioning is never that quite that simple, and we have had to make a few tweaks to the system to make sure it didn't cause subtle and difficult-to-debug problems. Three of these problems, which we call the "bad update" problem, the "magic upgrade" problem and the module state problem, deserve special attention.

The bad update problem

If a program relies on buggy, undocumented or otherwise subject-to-change behavior in a package (for instance because it works around a bug), then it may break in the presence of upgrades that PLaneT thinks of as compatible (for instance the eventual bug fix). We expect these cases to be the exception, not the rule (if we thought these were the common case then we wouldn't have added versioning to PLaneT at all), but they could still represent major problems for programmers unlucky enough to have to deal with them.

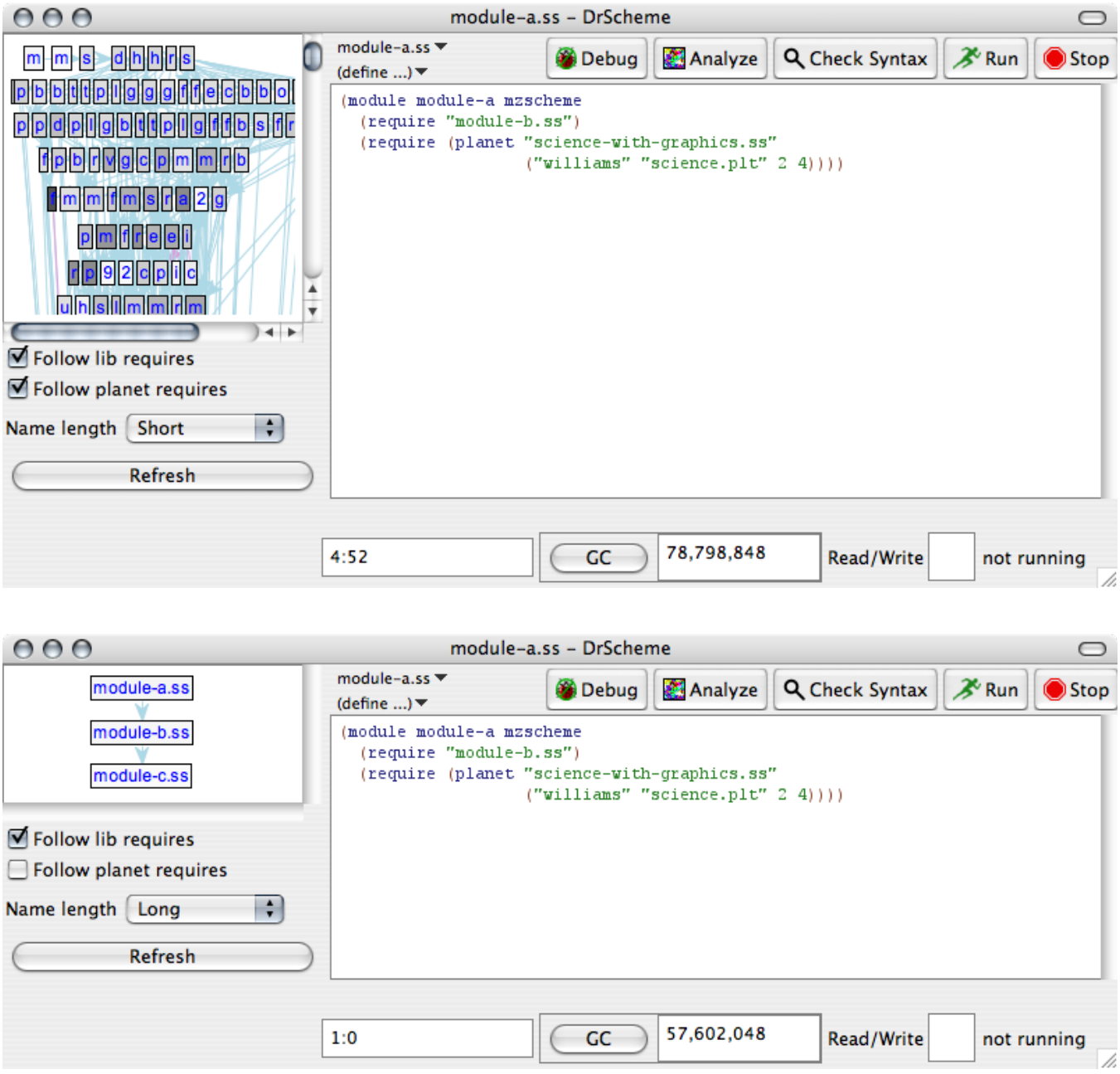

Figure 2. Sometimes special cases are important: the module browser displaying all requirements (above), and hiding PLaneT requirements (below)

To help those programmers cope, we decided early on to follow the lead set by the other component deployment systems we studied and make it possible to write a PLaneT **require** statement that asks for a more specific package than just anything backwards-compatible with a named package. In general, a PLaneT require specification may specify any range for a package's minor version (but not the major version, since two different major versions of a package don't have the same interface in general); for instance the package specifier ("soegaard" "galore.plt" 3 (0 1))) indicates that either package version 3.0 or version 3.1 is acceptable, but no others. In fact the normal way of writing a package specifier — ("soegaard" "galore.plt" 3 0) — is just syntactic sugar for ("soegaard" "galore.plt" 3 (0 +inf.0)). Similarly, the equality form ("soegaard" "galore.plt" 3 (= 0)) is just syntactic sugar for ("soegaard" "galore.plt" 3 (0 0)). We hope that programmers will not need to use these more general facilities often, but we expect that they are occasionally very useful to have around.

The magic upgrade problem

A subtler and stranger behavior that we had to be careful not to allow in PLaneT is what we call the magic upgrade problem, in which a user could run a program and end up changing the behavior of seemingly unrelated programs. If, in satisfying a requirement for a particular package specification, the PLaneT client always were to always look for the most recent locally-installed package that meets the criteria, then the following situation could arise: suppose a user has a program P that requires package A with package version 1.0 or later as part of its implementation, and has package A version 1.1 installed locally to satisfy that requirement. If package A releases a new version, 1.2, and then the user runs program Q which requires package A version 1.2 or later, then PLaneT must install package A version 1.2 locally. But now, if the user runs the original program P, *its* behavior will change because instead of using version 1.1, the client can now supply it with the newer package version 1.2. This might change program P's behavior in unpredictable ways, which is bad because according to our design principles, the user isn't supposed to have to know that package A even exists, much less that P and Q happen both to use different versions of it and so running Q might "magically upgrade" P without any warning!

Rather than allow this to happen, we have introduced a layer of indirection. When a program requires a PLaneT package for the first time, the PLaneT client remembers which particular package it told mzscheme to use to satisfy that requirement. Whenever the same program asks again, it returns the same path, regardless of whether newer packages are available. We call these associations "links" and a particular module's set of links its "linkage"; the links are collectively stored in the PLaneT client's "linkage table." The PLaneT client adds a link to a module every time it resolves a request for that module, and a module's linkage persists until the user explicitly clears it.

Module state conflicts

Another problem that can come up when two versions of a package are installed at the same time is that they may inadvertently interfere with each other because they both behave in similar ways but have different module state. In PLT Scheme, a module has only one copy of its mutable state, no matter how many times other modules require it — this allows a module to establish its own whole-program invariants or to regulate access to unique, non-duplicatable resources on the system. For instance, suppose a library author writes a module that maintains some internal state in the course of its duties:

```
(module db mzscheme

  (define *dbfile* "my-database.db")

  (define num-items 0)


  (define (write-to-db datum)
    (with-output-to-file *dbfile*
      (lambda ()
        (display datum)
        (set! num-items (add1 num-items)))))


  (provide write-to-db))
```

The author of this code might reasonably expect that no matter how a program used the *db* module to write values into my-database.db, *num-items* would remain a faithful count of the number of items there (assuming that the file started out empty, of course). That author would be correct, because no matter how many modules require the *db* module they will each get the same copy with shared internal state.

If the author puts this module in a PLaneT package, the same behavior applies. However, that behavior may not be good enough anymore, because PLaneT allows multiple versions of the same package to run side-by-side. Suppose the developer releases a new version of the *db* package:

```
;; new db module for PLaneT package version 1.1
(module db mzscheme

  (define *dbfile* "my-database.db")
  (define num-items 0)
  (define (write-to-db datum) ... [as before])
  (define (read-from-db)
    (with-input-from-file *dbfile*
      (lambda ()
        (do ((ret null)
             (i 0 (+ i 1)))
            ((= i num-items) ret)
          (set! ret (cons (read) ret))))))


  (provide write-to-db read-from-db))
```

Again, the developer might expect that *num-items* is always a true count of the number of items written into my-database.db. But it might not be the case anymore: from the solutions to the bad update problem and the magic upgrade problem, we know that for a variety of reasons different libraries within the same program might end up loading the same package with two different versions because of packages insisting on particular package versions or because of different modules in the same program getting different links. In particular, that means that a single program might use the *db* modules from package version 1.0 and package version 1.1 at the same time, and as far as mzscheme is concerned those are two separate modules with completely distinct states. If that were to happen, writes that went through *db* version 1.0 would not be reflected in version 1.1's counter, possibly leading to a corrupt database file or even a program crash if the program called *read-from-db*.

We expect that most packages will not exhibit problems like this, because most programming libraries do not rely on invariants between module-level state and system state in the way the *db* module does. However, we also expect that for the modules that do rely on those invariants, this problem could easily be a source of inscrutable bugs. Therefore our design takes the conservative approach and by default we do not allow multiple versions of the same library to be loaded at the same time, unless explicitly allowed.

Considering all of these issues together, we have arrived at the following final policy for clients resolving a PLaneT require statement for a particular program and a particular package.

1. PLaneT first decides what package to use to satisfy the request:
   (a) If the program has a link in the linkage table to a particular version of the package being requested, then PLaneT always uses that package.
   (b) If the program does not have a link, then PLaneT obtains a package that satisfies the request either from its local cache or from the central PLaneT server.
2. If PLaneT decides on a version of a package, and another version of that same package is already loaded (not just installed but actually running) in some other part of the program, then PLaneT signals an error unless the package itself explicitly tells PLaneT that simultaneous loading should be allowed. Otherwise it uses the package it has decided on to satisfy the request.

8. Evaluation

Up to this point we have described PLaneT as we originally designed it. PLaneT has been in active use for over two years, and in that time we have gained a lot of experience with how well our design decisions have worked out in practice. Generally we have found that our design works well, though we have identified a few problems and oversights, both with its technical design and with its ease-of-use features (which we consider to be at least as important).

First the good: PLaneT has quickly become many developers' default way to distribute PLT Scheme libraries, and over the course of two years we have received 115 packages and nearly 300,000 lines of Scheme code, compared to about 700,000 of Scheme code in PLT Scheme's standard library (which includes among other things the entire DrScheme editor implementation). These user-contributed libraries have been very useful for adding important functionality that PLT Scheme itself does not include. For instance, PLaneT currently holds implementations of several sophisticated standard algorthims, three different database interfaces, two unit-testing frameworks, and several bindings to popular native libraries. PLT Scheme itself does not come with much built-in support for any of these, so the easy availability of PLaneT packages has been a big help to people writing programs that need those. (Schematics' unit-testing package SchemeUnit [16], packaged as schemeunit.plt, is the single most popular package on PLaneT and is required by over a quarter of the other available packages.) Its utility is also demonstrated by its usage statistics: PLaneT packages have been download 22475 times (as of the moment of this writing), an average of about 30 a day since PLaneT was launched (and an average of about 50 a day in the last year).

We have also found that integration of package installation into the **require** statement has had the effect we had hoped it would, that programmers would have fewer qualms about using packages than they would otherwise. Our evidence is anecdotal, but we have found that code snippets on the PLT Scheme users' mailing list and in other programming fora have frequently included PLaneT require statements, without even necessarily calling attention to that fact; before PLaneT it was not common at all for people to post code that relied on nonstandard libraries.

It is harder to say how successful our package version numbering strategy has been. We have not yet heard of any problems resulting from backwards incompatibilities (with one arguable exception as we discuss below), but we have also found that nearly half of all packages (54 out of 115) have only ever released one version, and most (72 of 115) had released only one or two versions. This is not in itself alarming — on CPAN, for instance, most packages only ever release one version as well — but none of our version-compatibility machinery is relevant to packages with only one version, so the fact that it has not given us problems is not very informative.

Another area where we are still unsure of whether our policy was correct is our choice not to perform any quality control on incoming packages. While we still believe that we cannot and should not try to verify that every submitted package is "good," we have begun to think that it may be beneficial to make a few sanity checks before accepting a package. For instance, if an author submits a new version of a package and claims the new version is backwards-compatible, we cannot verify that absolutely but we can at least make sure that it provides all the same modules the old package did, and that none of those modules fails to provide a name that was provided before. This does impose an additional hurdle for package authors to clear, but in practice it seems that the only packages that fail this test have obvious packaging errors that authors would like to know about.

There have been a few design problems we have had to negotiate. One significant problem was that as new versions of PLT Scheme were released, they introduced backwards incompatibilities, and packages written for one version did not work later ones. Similarly, code written for newer versions of PLT Scheme used features that were not available in older versions, which could cause problems for users of those older versions who tried to download the new packages. This is the same problem that PLaneT's version numbering scheme tries to address, of course, but since PLT Scheme is not itself a PLaneT package we could not reuse that scheme directly. Furthermore, the PLT Scheme distribution does occasionally introduce large, generally backwards-incompatible changes in its releases, big events that define a new "series", but much more regularly it introduces more minor incompatibilities. These incompatibilities are obscure enough that we thought PLaneT packages would nearly never be affected by them, so we did not want to make package authors release new versions of their packages in response to each one. Considering these, we decided on the following policy: when an author submits a package, we associate a PLT Scheme series with it (currently there are two such series, the "2xx" series and the "3xx" series), which places that package in the 2xx or 3xx repository. By default we assume that any PLT Scheme version in a series can use any package in its corresponding repository; packages can override this assumption by indicating a minimum required PLT Scheme version if necessary.

Another problem that quickly became apparent in our initial design was that we had imposed an annoying hurdle for package developers. Developers understandably want to test their packages as they will appear to users, and in particular they want to be able to require and test packages using the (**require** (**planet** $\cdots$)) form since that form is what others will use. With our original design, there was no way to do that; the only way to have your package accessible as via PLaneT was to actually submit it, so developers had to interact with their code using either a (**require** (**file** $\cdots$)) or a (**require** (**lib** $\cdots$)) statement instead. This caused many problems and considerable frustration. Our first attempt to solve the problem was to allow programmers to install a package file directly to their PLaneT clients' local caches without going through the server. This helped but did not eliminate the problem, since programmers still had to create and install a package every time they wanted to run their tests. Based on the response to that, we arrived at our current solution: programmers can create "development links" that tell the PLaneT client to look for a given package version in an arbitrary directory of the programmer's choice. Since we added development links we have not had any more problems with programmers not being able to test their packages adequately.

9. Conclusions and related work

With two years of experience, we feel confident now in concluding that PLaneT has basically met the goals we set out for it: it is easy to use for package authors, developers who use PLaneT packages, and end users who don't want to think about PLaneT packages at all. We attribute this ease of use to our unusual inclusion of package management features into the language itself, which has allowed us to do away with the concept of an uninstalled package from the programmer's perspective. While to our knowledge that feature is unique, PLaneT otherwise borrows heavily from previously-existing package distribution systems and language-specific library archives.

As we have already discussed, we have taken insight for our design from the designs of CPAN, Debian Linux's `dpkg` system and Ruby's RubyGems system; and there are several other package management systems that also bear some level of similarity to PLaneT. Many Linux distributions come with automatic package managers (such as Gentoo's `portage` system [15] and the RPM Package Manager [7]), and there have also been a few implementations of completely automatic application-launching systems, *i.e.* application package managers that eliminate the concept of an uninstalled application: Zero Install [8] and klik [9] are examples of this kind of system. Furthermore there are many language-specific component and component-deployment systems for other language implementations; examples of these include Chicken Scheme's eggs system [17], Michael Schinz' `scsh` package manager [13], and the Common Lisp `asdf` and asdf-install system [2].

We hope with PLaneT we have achieved a synthesis of all these ideas into a single coherent system that is as natural and as easy to use as possible. We believe that every programming language should make it simple for one programmer to reuse another programmer's code; and we believe a key part of that is just making sure that programmers have a standard way to access each others' programs. As we have shown, moving package management into the language's core is a powerful way to achieve that goal.

Acknowledgements

We would like to acknowledge Robby Findler, Dave Herman, Ryan Culpepper, Noel Welsh, and the anonymous reviewers for their valuable comments about PLaneT in general and this paper. Thanks also to the National Science Foundation for supporting this work.

References

[1] Elaine Ashton and Jarkko Hietaniemi. Cpan frequently asked questions. `http://www.cpan.org/misc/cpan-faq.html`.

[2] Dan Barlow and Edi Weitz. ASDF-Install. `http://common-lisp.net/project/asdf-install/`.

[3] Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In *Workshop on Scheme and Functional Programming*, 2004.

[4] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.

[5] Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://www.plt-scheme.org/software/mzscheme/`.

[6] Matthew Flatt. Composable and compilable macros: You want it *when?* In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

[7] Eric Foster-Johnson. *Red Hat RPM Guide*. Red Hat, 2003.

[8] Thomas Leonard. The zero install system, 2006. `http://zero-install.sourceforge.net/`.

[9] Simon Peter. klik, 2006. `http://klik.atekon.de/`.

[10] Matt Pietrek. Avoiding DLL hell: Introducing application metadata in the Microsoft .NET framework. *MSDN Magazine*, October 2000. Available online: `http://msdn.microsoft.com/msdnmag/issues/1000/metadata/`.

[11] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 2001.

[12] RubyGems user guide, 2006. `http://www.rubygems.org/read/book/1`.

[13] Michael Schinz. A proposal for scsh packages, August 2005. `http://lampwww.epfl.ch/~schinz/scsh_packages/install-lib.pdf`.

[14] Gustavo Noronha Silva. *APT HOWTO*, 2005. `http://www.debian.org/doc/manuals/apt-howto/apt-howto.en.pdf`.

[15] Sven Vermeulen *et al*. A portage introduction, 2006. `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1`.

[16] Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL: Two little languages. In *Proceedings of the Third Workshop on Scheme and Functional Programming*, 2002.

[17] Felix L. Winkelmann. Eggs unlimited, 2006. `http://www.call-with-current-continuation.org/eggs/`.