Generating dependency constraints between packages using static code analysis

Maëlick Claes, Tom Mens
Software Engineering Lab
maelick.claes@umons.ac.be, tom.mens@umons.ac.be

University of Mons

Abstract

When developing software components in a software ecosystem, using dependency relationships is a common practice to ease component management and maintenance. An important and well-known challenge is how to deal with these dependencies between components when the system evolves. Version constraints are often used to limit these dependency relationships in order to assure that each component will work with all its dependencies. Still, this information can be missing or inconsistent. The R ecosystem, composed of thousands of statistical software packages suffer from many maintainability issues and the lack of dependency constraints has been pinpointed by the community. In this presentation we present a work in progress technique based on static code analysis we used to detect dependency constraints between R packages and its current limitations.

1 Introduction

Managing software components using dependencies is very convenient both for users and developers. It allows users to easily install software and developers to easily reuse software written by others. However, when component repositories grow in size, these dependency relationships can become an issue. Problems such as broken components [4] or conflicts between components [10] can appear. Updating [11] a component can result in many other components breaking and heavily affect users.

Tools, such as *dist-check*[1], *coinst*[2] and *comigrate*[3] already exist to detect these problems and their causes. Even though these tools have been designed for Linux distributions, and Debian in particular, they can be easily extended for other package-based distributions. For that they need, among other, constraints on component versions associated to dependencies. When this information is missing or incomplete, they can become unable to detect some of the problems.

In previous work [5], we explored the ecosystem of R packages, and found that such packages are developed and distributed through a variety of platforms. For the distribution of R packages, *CRAN* is the official source, while for their development *GitHub* has become the most popular choice. We studied in more depth [6] the problems caused by inter-repository dependencies between *CRAN* and *GitHub*, and how it potentially impacts the community.

In [2], we studied the maintainability of *CRAN* packages in terms of errors discovered by the official R package check tool, and how this relates to package dependencies and package updates. We also studied the presence

[1]http://dose.gforge.inria.fr/
[2]http://coinst.irill.org/
[3]https://www.irif.univ-paris-diderot.fr/~vouillon/coinst/comigrate/

of identical function clones between R packages [3] in *CRAN*. Based on these insights we created a web-based dashboard for helping *CRAN* package maintainers to deal with such issues [1].

In this presentation we propose a work in progress technique, and its limitations, based on static code analysis to detect dependency constraints for R packages.

2 The R ecosystem

There are many popular languages, tools and environments for statistical computing. On the commercial side, among the most popular ones are SAS, SPSS, Statistica and Stata. On the open source side, the R language and its accompanying software environment for statistical computing (`www.r-project.org`) is a strong competitor, regardless of how popularity is measured [8].

R forms a *component-based software ecosystem*. Its package management system provides an easy way to install third-party code and datasets alongside tests, documentation and examples[7]. The main R distribution installs a few *base* and *recommended* packages.

Thousands of additional packages are developed and distributed through different repositories. *CRAN*, the *Comprehensive R Archive Network* (see `cran.r-project.org`), constitutes the official R repository offering both source and precompiled stable packages compatible with the latest version of the R environment. Getting one's package accepted on *CRAN* can, however, be a painful process, due to the strict quality policy imposed by *CRAN*. In addition, the large number of available non-archived packages (> 8,000 in April 2016) is becoming a bottleneck [7] and leads to package dependency problems: *"the number of packages on* CRAN *and other repositories has increased beyond what might have been foreseen, and is revealing some limitations of the current design. One such problem is the general lack of dependency versioning in the infrastructure."* [9]

While many other R package repositories exist (*BioConductor*, *Omegahat*, *R-Forge*, etc), they are all much smaller than *CRAN*. But with the appearance of R packages such as *devtools* that allow the installation of a package from many different sources, and given the increasing popularity of *GitHub* as a platform for distributed software development, *GitHub* has become a major source of R packages. It has even exceeded *CRAN* in terms of number of hosted packages as shown in Fig. 1.

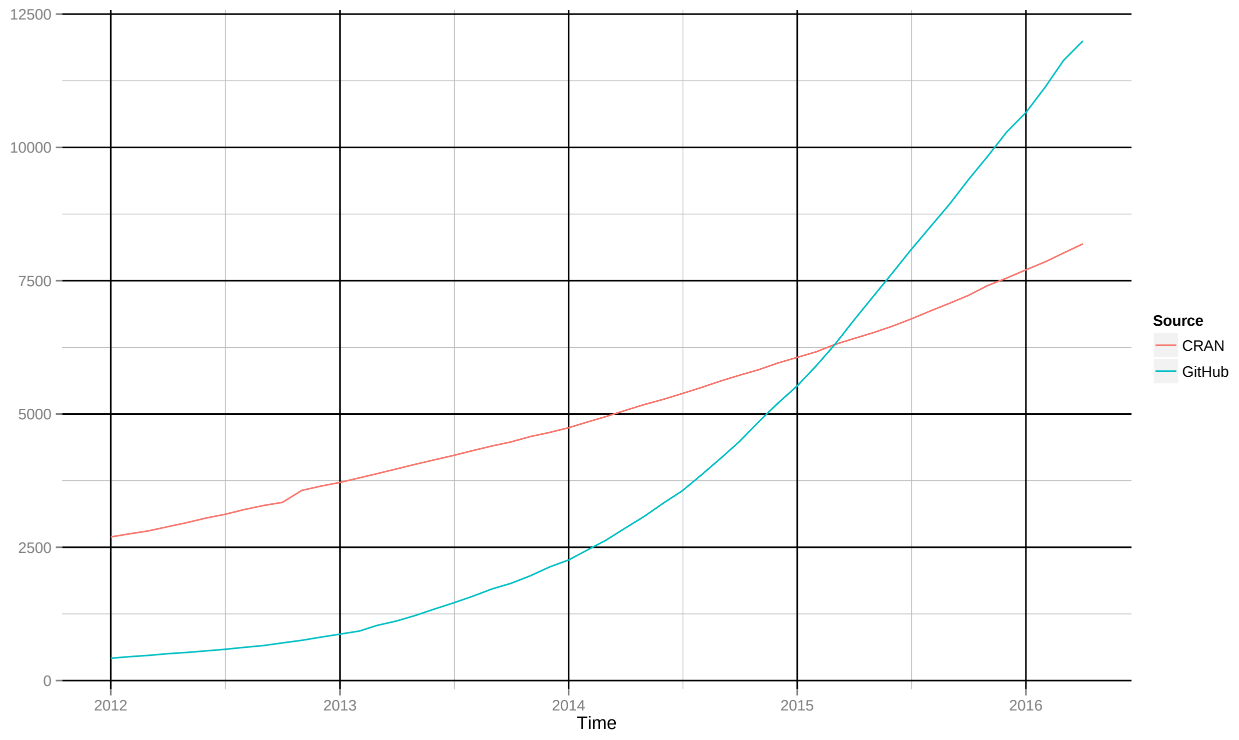

Figure 1: Evolution of the total number of R packages in *CRAN* and *GitHub*

Because of the general lack of dependency constraints, and the absence of a quality control like on *CRAN* where the latest version of all packages are tested together, knowing whether a *GitHub* package works with the current version of its dependencies is not an easy task. Thus it is important and useful for the community to be able to get this information. Moreover it would allow other existing tools, such as *dist-check*, to be used on R packages.

3 Extracting dependency constraints in R packages

In this section we first present the general structure of R packages and how R code can be reused by other packages using *NAMESPACE* files. Then we present our methodology for generating dependency constraints by analyzing R code found in packages.

3.1 Structure of R packages

An R package can contain different elements, ranging from code in R, or other language, data, documentation or tests. To detect incompatibilities between a package and some versions of its dependencies we focus on R code as it is included in the majority of packages.

While R code looks at first sight like an imperative language with some elements of object oriented programming, it is actually a functional language at its very core. All operators and syntax elements are actually function calls and a piece of R code could easily be translated to a LISP inspired language. For example the following piece of code:

```
t <- read.csv("file.csv")
if (ncol(t) == 2) {
  t[[3]] <- t[[1]] + t[[2]]
}
```

Could be translated to a LISP inspired language as:

```
('<-' (read.csv "file.csv"))
(if ('==' (ncol t) 2)
  ('{' ('<-' ('[[' t 3) ('+' ('[[' t 1) ('[[' t 2)))))
```

In order for packages to reuse code from other packages, each package must define a *NAMESPACE.* This file contains instructions about which global variables and functions it imports and exports. Those exported are the ones defined in the package and that can be called by other packages. Those imported are the ones exported by dependencies that are available to be used by the package importing them.

The following excerpt from the *NAMESPACE* file of package *abc 2.1* shows the different imports and exports of the package. Multiple functions are exported (*abc*, *cv4abc*, etc), a few functions are imported selectively (*plot* and *points* from package *graphics*, *nnet* and *multinom* from package *nnet*, etc), and all functions and variables declared in package *abc.data* are imported.

```
export(abc, cv4abc, postpr, cv4postpr, expected.deviance, gfit, gfitpca)
importFrom(graphics, plot, points)
importFrom(nnet, nnet, multinom)
importFrom(MASS,lm.ridge)
importFrom(quantreg,rq)
importFrom(locfit,locfit)
import(abc.data)
```

Additionally a package can also export functions using a regular expression. For example the instruction `exportPattern("^[^\\.]")` export every global function defined in the package and that do not start with a dot.

3.2 Methodology

One of the simplest checks to ensure a piece of R code works is to ensure that all of the functions called inside the package are defined somewhere. By checking function calls between packages, we can identify which versions of a dependency are required in order to satisfy all the function calls to that package.

For a given package, in order to determine which versions of its dependencies are compatible, we:

1. parse the R code of the package and its dependencies;
2. identify all variables and functions defined at a global level inside the package;
3. identify all function calls in the checked package that do not refer to a variable in its local scope;
4. identify all functions exported inside dependencies;
5. identify all functions imported by the package from its dependencies;
6. match function calls to imported functions from other packages.

The matching between function calls and imported functions allows us to identify all function calls that are not defined by any package, or specific package versions.

We can also detect more advanced problems such as potential conflicts between packages. While one can import functions selectively from a dependency, it is common to import all functions. When a package depends on two other packages and imports every functions from at least one of them, this can cause some problems if in the future one of them define a function with the same name as a function of the other package.

However the technique has a few pitfalls. R packages can import and export both functions and regular values. Because this can only be determined at runtime, we consider all exported objects as potential functions. This can create false positive when matching function calls with variables instead of functions.

Moreover we identified global variable and function definitions using the most commonly used assignment operators (`<-` and `=`). It is however possible to bind a value dynamically. This prevents us from detecting such functions. For example package *R.oo*[4] defines multiple functions using the *attach* function and export them using regular expressions.

4 Conclusion and future work

We proposed a technique based on static code analysis to detect incompatibilities between an R package and specific versions of its dependencies. This can be used to generate constraint on these dependencies. We also presented a few pitfalls of the approach caused by the dynamic nature of the R language. Future work includes validating our technique on *CRAN* packages in order to asses the importance of these pitfalls. Using our technique and a tool such as *dist-check*, a large scale study could be conducted on the evolution of dependency incompatibilities in both *CRAN* and *GitHub* R packages.

References

[1] Maëlick Claes, Tom Mens, and Philippe Grosjean. maintaineR: A web-based dashboard for maintainers of CRAN packages. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, pages 597–600, 2014.

[2] Maëlick Claes, Tom Mens, and Philippe Grosjean. On the maintainability of CRAN packages. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, pages 308–312, 2014.

[3] Maëlick Claes, Tom Mens, Narjisse Tabout, and Philippe Grosjean. An empirical study of identical function clones in CRAN. In *Int'l Workshop on Software Clones (IWSC)*, pages 19–25, 2015.

[4] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in FOSS distributions: details and challenges. *CoRR*, abs/0902.1610, 2009.

[5] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. On the development and distribution of R packages: An empirical analysis of the R ecosystem. In *European Conf. on Software Architecture Workshops (ECSAW)*. ACM, 2015.

[6] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[4] `https://cran.r-project.org/web/packages/R.oo/index.html`

[7] Kurt Hornik. Are there too many R packages? *Austrian Journal of Statistics*, 41(1):59–66, 2012.

[8] Robert A. Muenchen. The popularity of data analysis software. http://r4stats.com/articles/popularity/, 2015.

[9] Jeroen Ooms. Possible directions for improving dependency versioning in R. *R Journal*, 5(1):197–206, June 2013.

[10] Jérôme Vouillon and Roberto Di Cosmo. On software component co-installability. *ACM Trans. Software Engineering and Methodology*, 22(4):34, 2013.

[11] Jérôme Vouillon, Mehdi Dogguy, and Roberto Di Cosmo. Easing software component repository evolution. In *Int'l Conf. Software Engineering (ICSE)*, pages 756–766, 2014.