Stefan Theußl and Uwe Ligges and Kurt Hornik

Prospects and Challenges in R Package Development

Working Paper

Prospects and Challenges in R Package Development

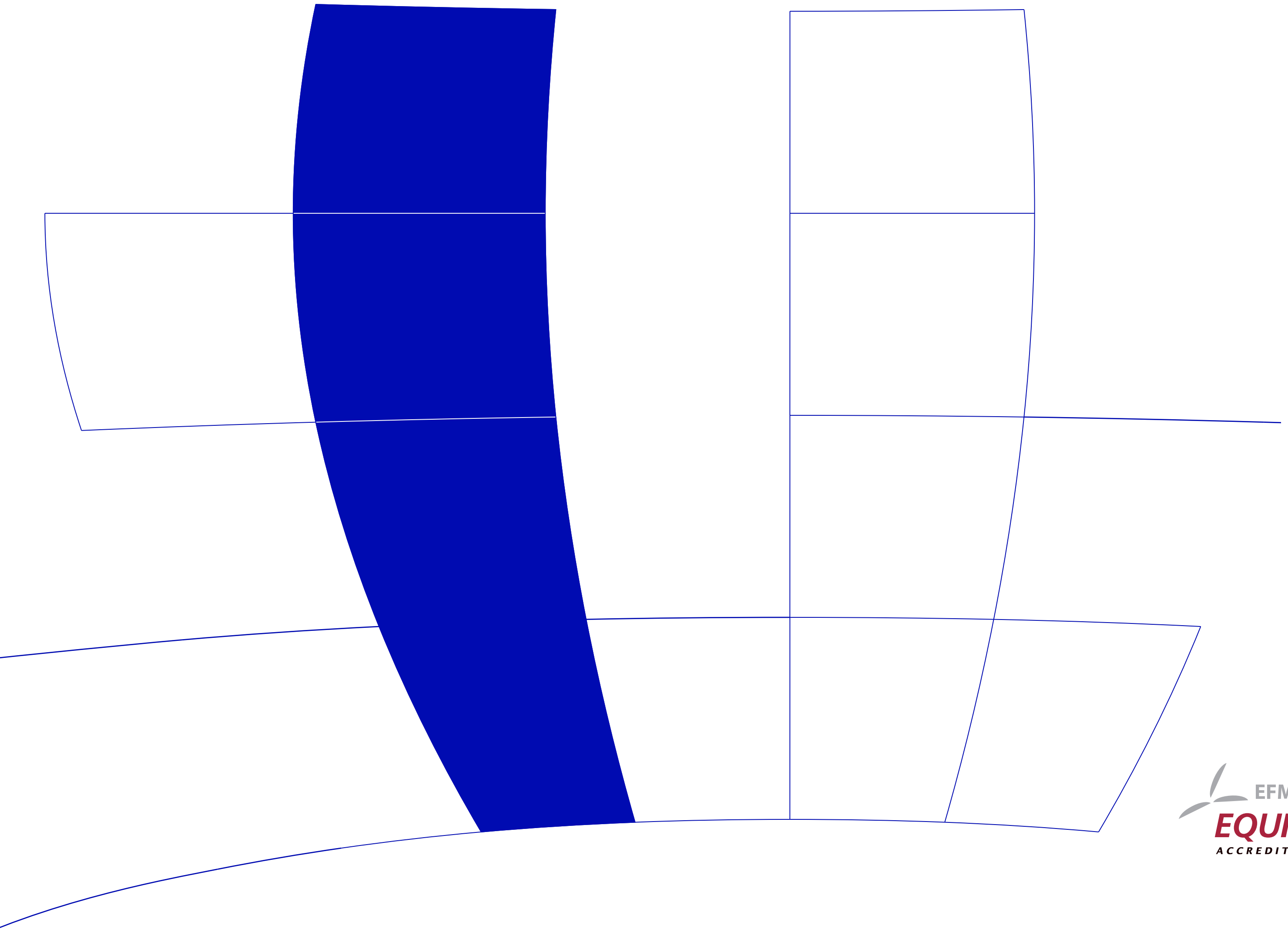
Stefan Theußl, Uwe Ligges, Kurt Hornik

Institute for Statistics and Mathematics
WU Wirtschaftsuniversität Wien

Research Report Series

Report 102
June 2010

http://statmath.wu.ac.at/



Prospects and Challenges in R Package Development

Stefan Theußl[1] Uwe Ligges[2] Kurt Hornik[1]

23-06-2010

[1] Institute for Statistics and Mathematics, WU Wirtschaftsuniversität Wien, Austria
[2] Department of Statistics, Technische Universität Dortmund, Germany

Abstract

R, a software package for statistical computing and graphics, has evolved into the lingua franca of (computational) statistics. One of the cornerstones of R's success is the decentralized and modularized way of creating software using a multi-tiered development model: The R Development Core Team provides the "base system", which delivers basic statistical functionality, and many other developers contribute code in the form of extensions in a standardized format via so-called packages. In order to be accessible by a broader audience, packages are made available via standardized source code repositories. To support such a loosely coupled development model, repositories should be able to verify that the provided packages meet certain formal quality criteria and "work": both relative to the development of the base R system as well as with other packages (interoperability). However, established quality assurance systems and collaborative infrastructures typically face several challenges, some of which we will discuss in this paper.

1 Introduction

R (R Development Core Team, 2010a), the prime open source environment for statistical computing and graphics, is nowadays present worldwide in many fields of applications within business, education, and research. Since more than a decade the R Development Core Team (R Core) has been responsible for the sustainable improvement and advancement of the *base system* of R, which delivers basic (statistical) functionality. It consists basically of the R interpreter, the *base* packages (base, stats, graphics, etc.), and several other packages which are recognized as *recommended* (Hornik, 2010). To enrich this basic infrastructure with additional functionality people located all around the world contribute source code in the form of extensions, so-called *packages*, which meet certain formal quality criteria.

When we think of packages as "products" and potential users as "customers" then the Comprehensive R Archive Network (CRAN, `http://CRAN.R-project.org`)—the prime repository for R related material—and other package repositories can be seen as warehouse-like storage areas from which products can be delivered. Modern views are driven by the understanding that customers are extremely heterogeneous with individual needs and preferences. 2417 packages (June 09, 2010) are available from CRAN reflecting many different areas of interest in the community. If the "right" product is not available, then it is rather easy for customers to become contributors by creating a new package possibly filling a gap. This decentralized and modularized way of creating software not only for the statistics but also a broader community is one of the reasons why the language R has become increasingly successful.

However, there are fundamental differences between the coordinated implementation of monolithic code in a software project on the one hand and a rather loosely coupled system of community contributions in a multi-tiered environment on the other hand. Looking at the fundamental development structures we find that there are several challenges to face but also prospects to see.

In this paper we describe the current state of contribution and distribution channels by providing products in repositories, see Section 2. Our main goal is to present corresponding challenges (cf. Section 3) we see in R package development and their implications on possible solutions. Furthermore, we discuss related aspects of package development in the R community including collaborative infrastructure and other services that support such a loosely coupled development model.

The original publication is available at `http://www.springerlink.com` (see Theußl et al., 2010).

2 Contribution and Distribution Channels

2.1 Products

Besides the R base system managed by R Core there are many packages available which have been developed and contributed by the R community. Packages are *standardized* entities that allow for easy distribution. They typically include meta-information (e.g. in the '`DESCRIPTION`' file), R code and corresponding documentation, possibly foreign code to be compiled/dynamically loaded (C, C++, FORTRAN, etc.), or interpreted (Perl, Tcl, Shell, etc.), as well as package-specific tests, data sets, demos, etc. In contrast to the rather monolithic code provided with the base system, the typically modularized code contained in packages does not necessarily work without functionality featured in other packages. Rather than reinventing the wheel, many package authors wisely reuse code of other packages that already exist. This means that new packages might depend on other packages that again depend on some other packages. Thus, there is a hierarchy of dependencies that could be broken by a simple bug in one of the packages. According to R Development Core Team (2010b) different "levels" of *dependencies* are declared in the '`DESCRIPTION`' file:

Depends: package B depends on functionality in package A in such a way that package A is loaded in advance of B,

Imports: package B imports (parts of) the namespace of package A into its own namespace,

LinkingTo: package B links to compiled code in package A,

Suggests: some functionality or examples in the documentation of package B depend on package A,

Enhances: package B enhances packages A functionality but works without A being present.

For illustration purposes suppose that B requires A to be loaded before B, i.e., the main functionality of B is only given if A is loaded, then this dependency of B on A has to be declared as *Depends.* Packages that have to be installed in order to successfully complete runtime checks have to be declared at least as dependency level *Suggests*, e.g., if functionality of A is used in the examples of B or called from a function in B. A package B declared as *Enhances* in A tells a user that functionality is provided by B that could enhance functionality of A given the latter is installed. Thus, whereas *Depends* implies the strongest relationship between packages, *Enhances* implies the weakest.

As a result some of the products cannot be used without acquiring other products.

2.2 Repositories

How do customers get their products? Almost all publicly available packages are usually downloadable from "standard" repositories like CRAN, Bioconductor, and R-Forge. But customers still have to be aware of the differences in the nature of the available products depending on where the products have been delivered from.

CRAN is the prime repository for R related material, i.e., documentation, binaries and sources of the base R distribution, and contributed packages. People usually submit their contributed source package via `ftp` upload and additional notification of the CRAN maintainers (`CRAN@R-project.org`). Generally, "incoming" packages are checked manually by the CRAN maintainers before they are included for provision. This ensures that (1) they are consistent with the guidelines specified in R Development Core Team (2010b) and (2) they are consistent in their dependencies to other code provided in the current state of the repository.

Bioconductor is the online repository of the Bioconductor Project (`http://Bioconductor.org`, Gentleman et al., 2004), an open development software project acting as a platform mainly for the creation of extensible software for computational biology and bioinformatics. It supplies the community with (peer-reviewed) software, data and meta data, papers and training materials. A unique feature of Bioconductor is the division of available packages into two main branches: a release branch and a development branch. Whereas in the former branch only bugfixes and documentation improvements are allowed, the latter is used for any other changes in the corresponding software. Twice a year—coordinated with new releases of the base R distribution—all Bioconductor packages in the development branch are approved collectively to move to the release branch after ensuring that they are coordinated and work well together.

R-Forge (`http://R-Forge.R-project.org`) offers a central platform for the development of R packages, R-related software and other projects (Theußl and Zeileis, 2009). In contrary to the software repositories mentioned above developers organize their work in so-called *projects*. Every project on R-Forge has various tools and web-based features for software development, communication and other services. Typically, a project hosts one or more R packages committed to a subdirectory called '`pkg`' of the project's SVN (Pilato et al., 2004) repository. Naturally, R-Forge offers a CRAN-style repository of R packages derived, i.e., built, from the committed source code. Usually, these packages have to be seen as development releases due to their volatile character. They directly reflect the development progress made in the repository.

All of the repositories mentioned above offer a QA system for contributions based on `R CMD check`. Later on, the packages are also regularly and automatically checked under various platforms. The results are usually made publicly available. Packages passing the checks are also provided in binary form, at least for Mac OS X and Windows.

Of course there are many other repositories typically with an emphasis on a narrower subject area available. E.g., the Omega Project for Statistical Computing (`http://www.omegahat.org`), or *Omegahat* for short, is a joint project with the goal of providing open source software (OSS) with a focus on web-based software, Java, the Java virtual machine, and distributed computing.

2.3 Other OSS Repositories

Many major OSS projects offer similar contribution and distribution channels like the R project. These projects include not only high-level programming languages like Perl or Python but also distributors of the Linux operating system like Debian (`http://www.debian.org`), Red Hat (`http://www.redhat.com`) or openSUSE (`http://www.opensuse.org`), and many more. E.g., the Debian project maintaining the popular operating system *Debian GNU Linux* offers a very sophisticated package management system (based on **dpkg**) and pursues a very similar approach as outlined in the above sections. Relationships between Debian packages are declared in the package's '`control`' file (Depends, Recommends, Suggests, Enhances, etc.) and packages are checked for certain formal quality criteria outlined in Jackson et al. (2010).

Although many popular OSS projects host their packages in a mirrored repository—e.g., Perl's packages can be obtained from CPAN (Ashton and Hietaniemi, 2007)—it is very hard to identify OSS projects which are as diverse in terms of infrastructure for collaboration and quality assurance as the R project. This diversity in the nature of the products and repositories may lead to challenges for developers and repository maintainers as is described in the following section.

3 Challenges

3.1 How can Interoperability be Achieved?

Packages provided in the standard repositories may depend on others, or suggest other packages to be used together with the submitted application. This has implications on the interoperability between packages. One may think of a "dependency structure" where package A depends on some functionality in another package B. Let us denote this formally as $B \in d(A)$, where $d(A)$ entails all dependencies of A. Suppose a new version of package B is released. Then one obviously has to verify whether the code in package A still operates with the changes introduced in B. Evidently this has to be done for each of the *reverse dependencies* of B denoted $d^{-1}(B)$, i.e., all packages depending on B. Furthermore, we also have to consider *recursive dependencies* of a given package A denoted by $d_R(A)$, i.e., the transitive closure of

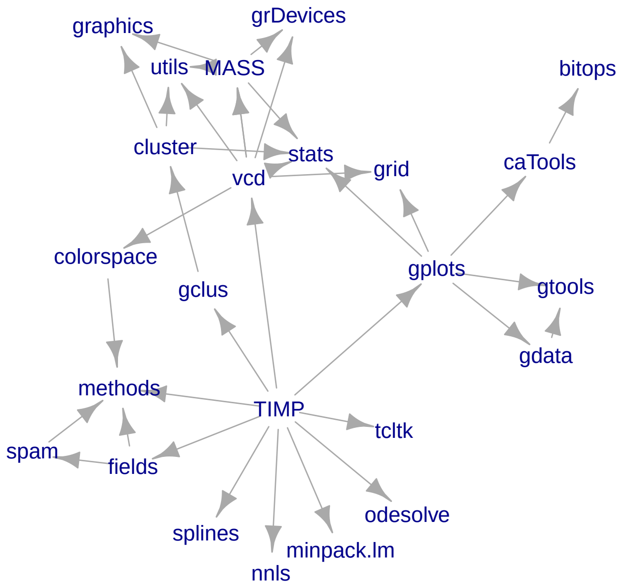


Figure 1: Recursive dependencies of CRAN package **TIMP**, d_R(TIMP) (based on dependency levels *Depends*, *Imports* and *LinkingTo*)

all dependencies of A. Thus, if $B \in d_R(A)$ and $C \in d_R(B)$ then $C \in d_R(A)$. From a computational point of view this poses challenges in resolving such reverse dependency structures since all the reverse dependencies, including recursive reverse dependencies (e.g. for a package $A \in d^{-1}(B)$ and $C \in d(B)$ we also know that $A \in d_R^{-1}(C)$), have to be considered for re-checking interoperability.

With the (almost exponential) growth of CRAN it has become more frequent that updates of an arbitrary package P lead to malfunction of one of its $d^{-1}(P)$ or even $d_R^{-1}(P)$. Recently, reverse recursive dependency checks have been introduced on CRAN and it is quite surprising that unrecognized problems have not been harmful before. In addition to regular checks, it might be necessary to provide updated (i.e., recompiled) binary versions of all $d_R^{-1}(P)$ as well, in particular if certain kinds of definitions (such as S4 classes) or linked (to be compiled) code are involved.

As indicated in Section 2.1 package maintainers specify dependencies in the ‘`DESCRIPTION`’ file. The dependency levels *Depends*, *Imports* and *LinkingTo* must be fulfilled at installation time of a package, and, additionally, packages declared as *Suggests* must typically be available when a package is checked. Thus, usually two different types of dependency graphs have to be calculated. First, graphs have to be calculated needed for the generation of the correct installation order of the packages. Figure 1 shows such a graph for the CRAN package **TIMP**. Note that the dependency levels *Depends*, *Imports* and *LinkingTo* are used in the examples, figures and tables for *non*-reverse (recursive) dependencies, whereas the level *Suggests* is added to these levels for reverse (recursive) dependencies. Second, graphs representing the recursive reverse dependencies of a package are needed in every new check attempt of the given package. Figure 2 shows such a graph for the CRAN package **clue** (all graphs have been created with the help of package **igraph**, Csardi and Nepusz, 2006).

The function `.package_dependencies()` from package **tools**, introduced with R-2.10.0, can be used to retrieve all relevant information for constructing the graphs of involved (reverse) dependencies of packages. Table 1 shows the binary dependency matrix of all CRAN packages: Although more than 800 packages do not depend on any other package and more than 1600 packages are not used by another package, we observe that many other packages show rather complex dependency structures. E.g., package **Metabonomic** depends (recursively) on 33 packages, or package **survival** has 1141 recursive reverse dependencies on CRAN (see Table 2). These extremely long lists of recursive reverse dependencies cause problems given the computational effort of re-building and re-checking all involved packages. Especially, if several package

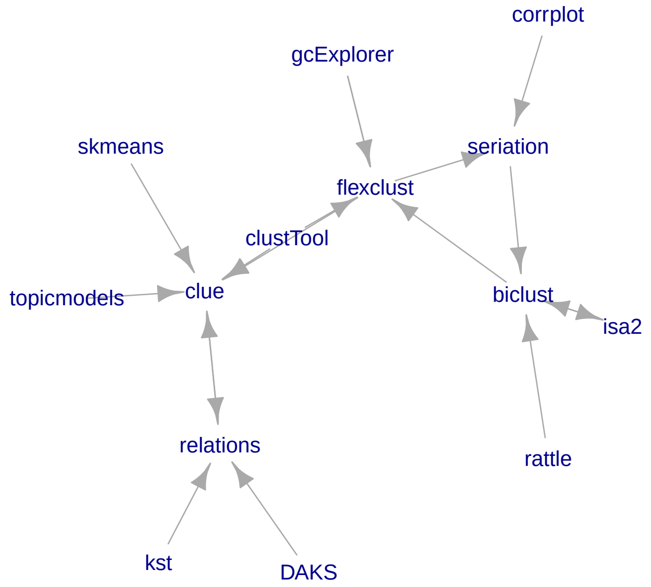


Figure 2: Recursive reverse dependencies of CRAN package **clue**, d_R^{-1}(clue) (based on dependency levels *Depends*, *Imports*, *LinkingTo* and *Suggests*)

Table 1: Number of CRAN packages with 0, 1, ..., and max. number of (recursive / reverse) dependencies (all calculations based on dependency levels *Depends*, *Imports* and *LinkingTo*, plus *Suggests* for flat and recursive reverse dependencies)

dependencies	0	1	2	3	4	5	6	max
flat	809	573	365	227	180	97	57	19
recursive	809	265	182	126	93	120	110	33
flat reverse	1614	336	132	83	33	37	20	313
recursive reverse	1614	245	106	66	52	25	20	1141

Table 2: Selected CRAN packages with extreme number of (recursive / reverse) dependencies (all calculations based on dependency levels *Depends*, *Imports* and *LinkingTo*, plus *Suggests* for flat and recursive reverse dependencies)

dependencies	MASS	survival	Metabonomic	sisus
flat	4	4	19	18
recursive	4	4	33	31
flat reverse	313	127	0	0
recursive reverse	1065	1141	0	0

updates occur on a day, it is impossible to catch up with the development without allowing for parallel checks and installations.

3.2 How can Growth be Handled?

The total number of available R packages is steadily increasing. Depending on combinations of different flavors of R (i.e., the branches *devel*, *patched*, and *release*), platforms (usually Linux, Mac OS X, Solaris, Windows), and architectures (SPARC, ix86, x86_64) building and checking of packages can become rather time consuming. As described before, even reverse recursive dependencies need to be checked on these different flavors of R on several platforms in order to provide reliable check results. Since the development version of R changes over time, regular checks (up to daily) of all packages of a repository are desirable.

For example, roughly 50 CPU hours on a modern (single-core) CPU are required to install (5.8 hours) and check (44.2 hours) all installable out of 2417 CRAN packages for 32-bit R-2.11.1 on Windows Server 2008 (x86_64). See `http://CRAN.R-project.org/web/checks/check_timings.html` for more information on actual installation and check timings on CRAN. This shows that it is impossible to deliver meaningful check results in a reasonable amount of time. Since it is desirable to get check results for development processes early, we obviously need a build and check system that finishes at least within 24 hours for each flavor of R in order to provide the check results when needed and make binaries available in time during an R release cycle.

As a possible solution parallel package installations and parallel checks have been introduced in R-2.9.0. E.g., user level parallel installations from source packages are possible in the function `install.packages()` which gained an additional argument `Ncpus = getOption("Ncpus")` to specify the number of packages to be installed in parallel. To make this possible, the package dependency structure is calculated in R, written to a '`Makefile`', and finally resolved by `make` (Stallman et al., 2002). The latter ensures that all $d(B)$ are in place before the installation process of B is started. Once all packages have been installed, one can check packages easily in parallel (e.g. also controlled by `make`) in a repository maintainers' mode that allows to omit a second installation step and just includes the logfile of the previous installation.

This reduces the install and check time on the aforementioned machine with its eight CPU cores almost linearly from 50 (on a single core) to 6.4 hours—given a sufficiently fast hard disc array is connected.

3.3 How can the Release/Development Tension be Handled for Collaborative Infrastructures?

Since the advent of collaborative infrastructure for R package development we are confronted with new types of challenges. On the one hand such platforms offer many advantages from a technological (efficient source code management, web-based communication, etc.) but also a social networking point of view. On the other hand this implies that typically such environments feature two versions of packages, a release and a (current) development one. Depending on the underlying development model pursued this has different consequences.

In view of the products and repositories offered in the R world (see Section 2) we identified two models: (1) a milestone driven approach and (2) a rather loosely coupled open development model.

The former approach typically involves creating software provided as monolithic code which delivers a pre-specified (documented) set of features. E.g., the functionality of the base R system is defined in (R Development Core Team, 2010c), well tested, and the code base of the current "stable" branch does not change significantly over a specific period of time. Usually, a new version of R is released twice a year. Only bugfixes are provided between releases (the so-called "patched" flavor of R). Another example is the Bioconductor project. Here, all packages in the stable branch are well tested to work with each other. Although there are no formal contracting methodologies the creation of interoperable components is emphasized by the Bioconductor project (*designing by contract*).

CRAN and R-Forge contributions on the other hand are typically developed using a rather loosely coupled open development model. In this model developers submit their application typically by sending standardized archives of source code including documentation to CRAN or, alternatively, by providing the source code in the corresponding SVN repository of the project on R-Forge. Developers typically need not follow any design contracts except those given by the programming language and the package structure.

Furthermore, contributors are allowed to upload their code without a specific schedule. As long as the contribution delivers valid results in the quality verification step it is considered for provision. This has several implications.

Suppose for example that R-Forge may also host development releases of packages already available from CRAN (or other repositories) possibly with a higher version number. Let $d_{CRAN}(P)$ and $d_{R\text{-}Forge}(P)$ be the dependencies of a package P hosted on CRAN and R-Forge, respectively. How should we resolve $d(P)$ on R-Forge? Is it sufficient to resolve only $d_{R\text{-}Forge}(P)$? Presumably not, since many of $d(P)$ are not (yet) hosted on this platform. On the other hand $d_{CRAN}(P)$ may not include packages already available on R-Forge. Our current solution is to calculate the dependency structure in the following way: First all $d_{CRAN}(P)$ and then $d_{R\text{-}Forge}(P) \backslash d_{CRAN}(P)$ are installed. This allows that all $d(P)$ are available but forces the system to use $d_{CRAN}(P)$ with higher priority. This decision reflects the fact that CRAN packages are known to be stable and "work". However, in certain circumstances developers may want the system to use (some of) $d_{R\text{-}Forge}(P)$ instead. Suppose **vcd** $\in d($**TIMP**$)$ (see Figure 1) provides a new function for visualizing categorical data in the package on R-Forge. The maintainer of **TIMP** is now interested in using this function in one of the package's examples. How can this problem be solved? The answer is not yet clear. One possible solution is to depend on the version of the corresponding package hosted on R-Forge which is greater than the version number of the same package hosted on CRAN, thus using "`>=`" in the fields of the '`DESCRIPTION`' file described in Section 2.1. This approach seems to be promising but then the calculation of the dependency structure has to be modified to take such "development dependencies" into account.

4 Discussion

The prospects of such a loosely-coupled development approach are diverse, among them rapid development, diversity, alternative approaches facing different aspects of implementations such as speed vs. accuracy. But there are also many challenges we have to face. Many of them have solutions that have been implemented or are shortly before being implemented, such as (recursive / reverse) dependency calculations and parallelized checks. Unfortunately, there are also some open issues. Given that a package is updated, all its reverse dependencies are re-built for a binary repository and distributed through all the CRAN mirrors. These are sometimes more than 300 packages on a single day, while just a few of them really need to be re-built in binary form. The infrastructure that supports calculation of the necessity of a binary re-built is not yet in place but is planned to be implemented for one of the next releases.

One of the solutions for the distributed development approach of packages, namely R-Forge, has been described in the previous section. Another solution supports package developers who do not have Windows machines for building or testing purposes available. This special service called *win-builder* is available from `http://win-builder.R-project.org`. It allows to upload source packages and provides corresponding Windows binaries and check results after a short amount of time.

Beside all computational work, the automation, and parallelization, we should not forget that there is still a lot of manual work to do for maintainers of huge package repositories, for example: Maintaining and adapting the scripts themselves, maintaining the hardware of devoted machines, setting up repositories for new versions of R, handling errors that were not covered by the scripts, answering questions of developers and users, and notifying package maintainers about recently broken packages. In other words, quality management can be improved by moving as many tasks as possible from human to computational resources. This provides the maintainers with time to improve the quality management even more.

References

E. Ashton and J. Hietaniemi. *CPAN FAQ*, 2007. URL `http://www.CPAN.org/misc/cpan-faq.html`.

G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. URL `http://igraph.sf.net`.

R. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Mächler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, and J. Zhang. Bioconductor:

Open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):R80, September 2004.

K. Hornik. *The R FAQ*. R Foundation for Statistical Computing, Vienna, Austria, 2010. URL http://CRAN.R-project.org/doc/FAQ.

I. Jackson, C. Schwarz, et al. *Debian Policy Manual*, 2010. URL http://www.debian.org/doc/debian-policy/. version 3.8.4.0.

C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion.* O'Reilly, 2004. Full book available online at http://svnbook.red-bean.com/.

R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2010a. URL http://www.R-project.org. ISBN 3-900051-07-0.

R Development Core Team. *Writing R Extensions.* R Foundation for Statistical Computing, Vienna, Austria, 2010b. URL http://www.R-project.org. ISBN 3-900051-11-9.

R Development Core Team. *The R Language Definition.* R Foundation for Statistical Computing, Vienna, Austria, 2010c. URL http://www.R-project.org. ISBN 3-900051-13-5.

R. M. Stallman, R. McGrath, and P. D. Smith. *GNU Make: A Program for Directing Recompilation.* Free Software Foundation, 2002. URL http://dsl.ee.unsw.edu.au/dsl-cdrom/gnutools/doc/gnu-make.pdf.

S. Theußl and A. Zeileis. Collaborative software development using R-Forge. *The R Journal*, 1(1):9–14, May 2009. URL http://journal.R-project.org/2009-1/RJournal_2009-1_Theussl+Zeileis.pdf.

S. Theußl, U. Ligges, and K. Hornik. Prospects and challenges in R package development. *Computational Statistics*, 2010. doi: 10.1007/s00180-010-0205-5. URL http://www.springerlink.com. To appear.