On the Maintainability of CRAN Packages

Maëlick Claes, Tom Mens, Philippe Grosjean
Software Engineering Lab, COMPLEXYS Research Institute, University of Mons, Belgium
firstname.lastname@umons.ac.be

Abstract—**When writing software, developers are confronted with a trade-off between depending on existing components and reimplementing similar functionality in their own code. Errors may be inadvertently introduced because of dependencies to unreliable components, and it may take longer time to fix these errors. We study such issues in the context of the CRAN archive, a long-lived software ecosystem consisting of over 5000 R packages being actively maintained by over 2500 maintainers, with different flavors of each package depending on the development status and target operating system. Based on an analysis of package dependencies and package status, we present preliminary results on the sources of errors in these packages per flavor, and the time that is needed to fix these errors.**

I. Introduction

When writing software, developers may want to reuse code that has been written by someone else rather than writing it again. Developers can do this by directly copying the code or by depending on it (e.g., using a software library). However, when there is no coordination between the developers of dependent software components, maintainability problems may arise: components may cease to function correctly because of changes made to the component(s) they depend upon. This can become particularly problematic in large software ecosystems containing thousands of different components maintained by thousands of different maintainers, some of which being considerably less active and responsive than others.

The CRAN package repository[1], which is the primary source of packages used by the R community[2], is currently experiencing such problems. The number of packages is growing very rapidly and has reached a size of over 5000 packages, which is considered by some as "too many" [1]. In addition, problems with the dependency versioning system of R have been reported and possible directions for improvement have been proposed [2]. For these reasons, we are empirically analysing the CRAN ecosystem. Our goal is to understand the impact of errors spreading across dependent packages, and the influence of the target operating system on this. This information could be used to provide more objective information to R package maintainers about which packages are easier to maintain and to depend upon.

II. Related Work

Our research pertains to the empirical analysis of evolving software *ecosystems*. Many researchers have contributed to this domain [3], either by focusing on the business aspects [4] or by focusing primarily on the socio-technical aspects [5], [6]. This paper falls in the second category. We adhere to the definition by Messerschmitt [7] who defines a software ecosystem as "a collection of software products that have some given degree of symbiotic relationships." This is consistent with Lungu's definition [5] "a collection of software projects which are developed and evolve together in the same environment." An important difference between a software ecosystem and an individual software project is the size and the division and coordination of labour. Whereas for a single software project there is a specific goal that all contributors try to achieve, this is not necessarily the case for a software ecosystem. Different packages of the ecosystems may have different objectives, and the coordination between the different package maintainers may be limited. Therefore, specific mechanisms need to be put in place to ensure the coordination and coherence of the ecosystem and its community as a whole.

Recently, Adams et al. [8] have studied the evolution of the R software ecosystem, focusing on the difference between core packages and user-contributed packages. Our work differs from theirs, since we focus on the errors introduced in R packages. In addition, we study the effect of the operating system(s) targeted by a particular package.

Also related is the research by Di Cosmo et al. [9], [10], [11], [12], [13]. They focus on the evolution and upgrade of components installed from software repositories in presence of broken dependencies and package conflicts. The case study they have used is the Debian GNU/Linux distribution.

III. About R and CRAN

R is a GNU project providing a complete free statistical environment. It provides a package system where packages can contain code (mainly written in R, though C, Fortran and other languages are allowed too), documentation, scripts, tests, data sets, and so on. Those packages can also depend on other R packages which can be retrieved automatically and recursively.

CRAN is the Comprehensive R Archive Network used to store R packages using different HTTP and FTP mirrors. It has a history going back to the oldest available source release in 1997, is managed by the R core team and currently contains more than 5000 packages and libraries. While there are other R package repositories (e.g., Bioconductor and R-forge), CRAN is the official, oldest and biggest one.

CRAN has a strict policy[3] and packages have to pass a check before being accepted on the repository[4]. The checking

[1]http://cran.r-project.org
[2]http://www.r-project.org
[3]http://cran.r-project.org/web/packages/policies.html
[4]http://cran.r-project.org/web/checks/

process is regularly reapplied on the full set of packages in order to test if they still pass it. Indeed, if one package is updated, other packages depending on it may no longer work correctly if the functions they were relying on have changed.

Packages are checked for different *flavors*[5]. Each flavor is a combination of operating system, hardware, R version and compiler. Essentially, there are flavors for Linux, Windows, Solaris and MacOS X, subdivided into development, patch or release versions.

CRAN specifies on the web page of each package the list of reverse dependencies (i.e., packages that depend directly on it). Even if the CRAN policy recommends not to break these reverse dependencies when updating a package, occasional dependency breaks are unavoidable. Packages can also fail the check if a new R release introduces, changes or removes features. One such example is the removal of the underscore symbol (_) as assignment operator.

Packages are never removed from CRAN, but archived instead. When a new R version is released, maintainers have to take care that their packages still pass the check. If this is not the case and if maintainers do not promise to fix the problems rapidly, they will eventually be archived when the next non-minor R version will be released, unless maintainers solve the reported errors before a stated deadline. For example, release 3.1.0 archives packages containing errors while release 3.1.1 permits the presence of erroneous packages.

A major issue with R and CRAN is that different versions of the same package cannot be installed together and that archived packages cannot be installed automatically. As a consequence, if a package maintainer decides for various reasons to stop maintaining his packages, dependency errors will arise in other packages relying on it, creating a ripple effect in the ecosystem. This problem is known by the R community and some solutions have been proposed to address the problem [2]: the use of Linux-like distributions or a package version management system like NPM[6].

IV. Experimental Setup

As stated in the introduction, our goal is to gain a better insight in the characteristics that determine the maintainability of CRAN packages, by analyzing the errors introduced in them, and the time needed to fix these errors, for different flavors. This insight should allow us to provide tool support to help R package developers to decide whether or not they should rely on a given package.

To achieve this goal we will study three research questions:

RQ_1: How does the use of a specific *flavor* impact errors in CRAN packages?

RQ_2: What is the source of errors in CRAN packages, and how are these errors fixed?

RQ_3: How long does it take to fix an error?

To answer these research questions we extracted historical metadata for all current and archived CRAN packages, available online[7]. We automated the extraction and processing of packages by implementing a set of R tools, available online for replication purposes[8]. These tools download the CRAN source packages, extract their content and store the content of the metadata of each package (stored in a DESCRIPTION file) into an SQL database. A package description file contains important information such as package version, maintainer and author information, dependencies to other packages, and reverse dependencies from other packages. In this paper, we only use the package name, version number and dependencies. We only consider strong dependencies (stored in the *Depends:* and *Imports:* fields of the description file) as these are required to install and load a package.

Next to the metadata of each package version, we also extracted content of the CRAN package checks on a daily basis for a 3-month period (from September 4 till December 3, 2013 included). Unfortunately, we could not go back further in time, since the historical information of CRAN checks is not available online. We continue collecting new data about daily checks for our future research. The extracted package checks report the daily *status* of each package for each flavor: OK, NOTE, WARNING or ERROR. Packages with an ERROR status are the most relevant, as these are the ones that will be archived upon release of the next non-minor R version.

V. Preliminary Results

A. Answer to RQ_1

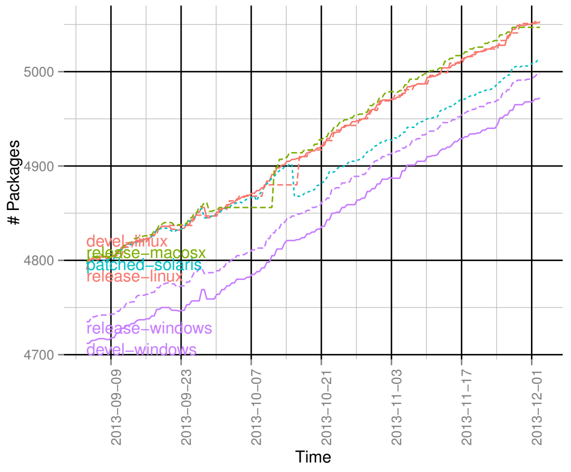
Because CRAN packages are checked for each *flavor*, we wish to know more about the differences between these flavors. Some operating systems ease package installation (e.g., by providing a repository for C libraries), thereby increasing the number of packages available for that particular flavor.

In this study, we decided to limit ourselves to the flavors relying on the released and development versions of R. We did not consider the MacOS X development flavor as it first appeared in November 2013 and thus it was not possible to compare its results to the other flavors. For the Linux flavors we only used the Debian distribution using the GCC compiler. For the Solaris flavor we used the R patched version since there is no release or development flavor available for this OS.

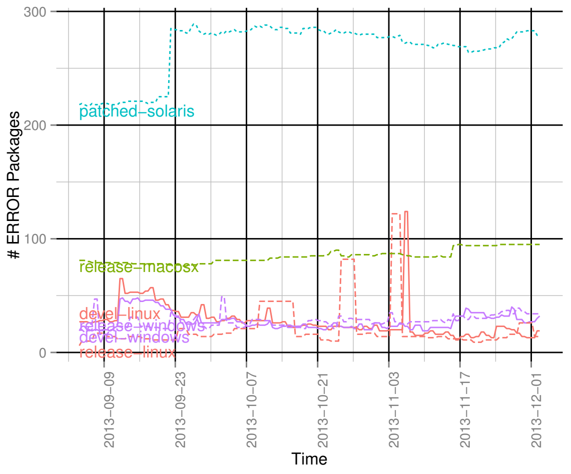
Fig. 1(a) shows the evolution of the number of active packages checked per flavor, based on our daily extracted data of CRAN package checks. Regardless of the considered flavor, we observe a rather linear growth of the number of packages over the 3-month period. For longer time periods, superlinear growths have been reported elsewhere [1]. We observe that Unix-based flavors have a similar number of packages. For Solaris, we observe a sudden drop in number of packages at half October. Windows-based flavors have a lower number of packages and also a different number of packages depending on the R version used. The observed difference compared to Unix flavors relates to the (un)availability of automatic installation and compilation tools. For example, the Debian

[5]http://cran.r-project.org/web/checks/check_flavors.html
[6]http://npmjs.org
[7]http://cran.r-project.org/src/contrib/
[8]http://github.com/maelick/extractoR

Linux distribution uses the APT tool to easily install external libraries. Windows doesn't provide such tools. Also, C code is generally compiled using a Makefile, making it more difficult to install on Windows than on Unix-based OS.



(a) Evolution of checked CRAN packages



(b) Evolution of CRAN packages with ERROR status

Fig. 1. Evolution of number of CRAN packages for each flavor. Full lines correspond to development versions, dashed lines are official release versions, and dotted lines represent testing (patched) versions.

Fig. 1(b) shows the evolution of number of CRAN packages with ERROR status. Solaris has more erroneous packages than the other flavors, as it is an uncommon OS for laptops or desktop PCs. MacOS X also has more ERROR packages than Windows and Linux flavors. This is probably because Windows is the most widespread OS and because Linux is freely available. It can be difficult for many developers to test and debug their packages for MacOS X. This may also explain the considerably lower variation ERROR packages for MacOS X than for Windows and Linux.

Table I summarizes the total number of errors introduced, the total number of errors that have been fixed and the total number of errors that have been both introduced and fixed, over the considered time period. It is worth noting that MacOS X and Solaris have more erroneous packages (cf. Fig. 1(b)) while less errors are fixed than for Windows and Linux. In particular for MacOS X, this confirms a less dynamic evolution of packages, with errors that remain unfixed or require a longer fixing time than we were able to measure. This means that in general packages are less well maintained on MacOS X than on Windows or Linux, implying that there either less R developers using MacOS X, or that these developers care less about the quality of their packages.

TABLE I
NUMBER OF NEWLY INTRODUCED ERROR STATUS, NUMBER OF FIXED ERROR STATUS AND NUMBER OF "FULLY RESOLVED" ERROR STATUS (ERROR STATUS THAT HAS BEEN INTRODUCED AND FIXED) DURING THE CONSIDERED TIME PERIOD FOR EACH PACKAGE.

Flavor	# new	# fixed	# full resolution
devel-linux	369	369	346
release-linux	358	342	337
devel-windows	262	244	233
release-windows	331	309	302
patched-solaris	200	127	93
release-macosx	50	32	20

The observed differences in number of ERROR packages across flavors implies that we should analyse flavors separately, rather than aggregating all results together.

B. Answer to RQ_2

The status of a CRAN package, as reported by the daily check, may change to an ERROR status, or may be fixed from ERROR to one of the other status (OK, NOTE or WARNING). In Table II, we identified four reasons for such a package status change: because the package itself gets updated (*PU*); because of the update of a dependency (*DU*), which can be subdivided into *direct* dependency update (*DDU*) and *indirect* dependency update (*IDU*); or due to external factors (*EF*).

TABLE II
TYPES OF CHANGES THAT MAY CHANGE THE STATUS OF A PACKAGE TO ERROR (OR THAT MAY FIX THE ERROR STATUS).

type	description
PU	Package Update: ERROR status change coincides with the release of a new package version
DU	Dependency Update: ERROR status change coincides with the release of a new package version of a dependency
DDU	*Direct* Dependency Update: ERROR status change coincides with the release of a new package version of a *direct* dependency
IDU	Indirect Dependency Update: ERROR status change coincides with the release of a new version of an *indirect* dependency
EF	External Factors: ERROR status change without a new package version or a new version of any dependency

We computed the CRAN package dependency graph each time the ERROR status of a package changed. For each of these packages we computed the list of all package dependencies. Table III and Table IV show, per change type, the absolute and relative number of ERROR status which got introduced (resp. fixed).

We observe that for the Windows release flavor the majority of errors are *introduced* at the same time as a dependency

TABLE III
ABSOLUTE AND RELATIVE NUMBER OF PACKAGE STATUS CHANGES TO ERROR (LISTED PER CHANGE TYPE)

Flavor	# *PU*	% *PU*	# *DDU*	% *DDU*	# *IDU*	% *IDU*	# *EF*	% *EF*
devel-linux	4	1.08	60	16.26	26	7.05	279	75.61
release-linux	7	1.96	80	22.35	15	4.19	256	71.51
devel-windows	20	7.63	68	25.95	4	1.53	170	64.89
release-windows	17	5.14	184	55.59	21	6.34	109	32.93
patched-solaris	74	36.63	51	25.25	8	3.96	69	34.16
release-macosx	40	80.00	0	0.00	0	0.00	10	20.00

TABLE IV
ABSOLUTE AND RELATIVE NUMBER OF PACKAGES IN WHICH THE ERROR STATUS IS FIXED (LISTED PER CHANGE TYPE)

Flavor	# *PU*	% *PU*	# *DDU*	% *DDU*	# *IDU*	% *IDU*	# *EF*	% *EF*
devel-linux	95	25.20	17	4.51	20	5.31	245	64.99
release-linux	57	16.29	53	15.14	24	6.86	216	61.71
devel-windows	75	30.12	8	3.21	3	1.20	163	65.46
release-windows	60	18.93	4	1.26	24	7.57	229	72.24
patched-solaris	95	67.38	6	4.26	2	1.42	38	26.95
release-macosx	21	58.33	1	2.78	0	0.00	14	38.89

package update (*DDU* or *IDU*) while errors are mainly introduced due to external factors (*EF*) for the other Windows or Linux flavors. For Windows and Linux, less then 10% of all errors are introduced at the same time as a package update (*PU*). This is because package updates are only allowed on CRAN if they do not contain errors on at least two operating systems. This also explains why there are still a few errors caused by a package update. 80% of all errors on MacOS X are introduced by a package update, probably because those packages have been developed by non MacOS X users. While Solaris contains errors introduced by *EF* or *DU*, it contains almost twice as many errors related to *PU* as MacOS X.

According to Table IV, less than 22% of all error statuses are *fixed* by a dependency package update (*DDU* and *IDU*). For Linux and Windows, the majority of errors (>61%) is fixed without any package update, but is due to external factors.

TABLE V
COLUMNS 2 AND 3: NUMBER AND PERCENTAGE OF ERROR STATUS INTRODUCED BY A *DU* AND FIXED BY A *DU* OR A *PU*. COLUMNS 4 AND 5: NUMBER AND PERCENTAGE OF ERROR STATUS FIXED BY A *DU* THAT WERE INTRODUCED BY A *DU*.

Flavor	# introduced	% introduced	# fixed	% fixed
devel-linux	63	81.82	27	75.00
release-linux	41	47.13	15	19.48
devel-windows	33	52.38	3	30.00
release-windows	30	15.46	4	14.81
patched-solaris	18	94.74	1	50.00

Table V shows per flavor how many errors have been introduced by *DU* and fixed by *DU* or *PU*. The percentage indicates their proportion compared to the total number of errors introduced (resp. fixed) by *DU*. For the Windows release flavor, less than 16% of the *DU* introduced errors have been fixed by *DU* or *PU*, implying that most of those errors are false positives. This may be due to the fact that *DU* can occur and that an error can still be introduced by *EF*.

Table VI shows, for errors fixed by a *PU*, how many were introduced by a *PU*, *DU* or *EF*, respectively. For Linux and Windows, the majority of ERROR status were introduced by a *DU*. From the observations we can conclude that errors reported by the CRAN check have different kinds of nature.

TABLE VI
NUMBER OF ERRORS THAT HAVE BEEN FIXED BY A *PU*, CATEGORIZED BY THE CAUSE OF THE INTRODUCTION OF THIS ERROR.

Flavor	# *PU*	# *DU*	# *EF*
devel-linux	1	36	29
release-linux	0	26	22
devel-windows	10	30	24
release-windows	11	26	11
patched-solaris	17	17	31
release-macosx	8	0	0

Depending on the operating system on which the check command is run, the number of errors is significantly different. For operating systems that are not popular among R developers (MacOS X and Solaris), the number of packages that contain errors will be higher than for common operating systems. The number of packages that contain an error when they are released will also be higher. For Linux and Windows, the majority of errors is unrelated to changes occurring in the depending packages. At the same time, the majority of errors fixed by the package developers were errors introduced by a change in the dependency. This means that errors caused by *EF* are either out of scope, out of control or irrelevant to developers. As a result, they need to focus their maintenance effort on fixing errors introduced by changes in depending packages. This may become detrimental to the maintainability of packages at the long term if the number of packages on CRAN keeps growing at the same pace.

C. Answer to RQ_3

Table VII summarizes the number of days needed to fix packages with errors for each flavor and error type. We observe that it takes longer to fix errors by releasing a new package version (*PU*) than for the other error types (*DU* and *EF*). For Windows and Linux, most errors fixed by *EF* have been fixed in one or two days while errors fixed by a *PU* take longer time to disappear. This means that, while there are many errors related to *EF*, many of them are fixed very quickly without intervention from developers. We also observe that errors on MacOS X need more fixing time than for other OS.

TABLE VII
NUMBER OF DAYS NEEDED TO FIX PACKAGES WITH ERROR STATUS, BY FLAVOR AND BY ERROR FIX TYPE

Flavor	Change type	Min	Mean	Median	Max
devel-linux	*PU*	1.0	9.03	6	48
	DU	1.0	3.67	1	47
	EF	1.0	1.37	1	16
release-linux	*PU*	2.0	9.16	5	47
	DU	2.0	4.74	3	46
	EF	2.0	2.99	2	16
devel-windows	*PU*	0.5	9.55	6	64
	DU	0.5	2.70	2	7
	EF	0.5	2.28	1	36
release-windows	*PU*	0.5	7.02	4	63
	DU	1.0	1.31	1	5
	EF	0.5	1.96	1	21
patched-solaris	*PU*	0.0	9.13	5	72
	DU	4.0	5.00	5	6
	EF	0.5	5.50	3	25
release-macosx	*PU*	1.0	15.58	13	32
	EF	1.0	7.42	8	14

VI. THREATS TO VALIDITY AND FUTURE WORK

Our results may be biased by limitations of the CRAN check tool and the other extraction and measurement tools that we have developed and used. A more important limitation is that, currently, we only have the CRAN package check data for a 3-month period. This is too short to be able to report any actionable results. We continue to collect results on a daily basis, in order to have a longer period of available data.

The analysis and results reported for *CRAN* may not be generalisable to other systems. We will study similar "archive networks" (like *CTAN* for TeX, *CPAN* for Perl, *CEAN* for Erlang, ...) in order to compare the maintainability of packages across archive networks, and to determine the main factors that can explain any difference found.

We plan to refine error types by looking at the package content. For example, through static analysis of the function dependency graph we could determine if a package update breaks package dependencies. We also wish to explore the role of package maintainers on the likelihood of errors in a package or its depending packages, as well as on the time needed to fix those errors. For example, errors might occur more often (or take longer to fix) if a depending package is maintained by a different person. Similarly, maintainers responsible of many different packages may be more "trustworthy" than others.

VII. CONCLUSIONS

With the aim to assess the maintainability of packages belonging to the CRAN archive network, we reported our early results on the analysis of errors introduced and fixed in these packages, over a 3-month period. We collected daily information from the CRAN website of the automated check of each package for each flavor.

We observed that some flavors are more error prone than others, mainly due to the targeted operating systems. We hypothesize that this is because many R developers pay less attention to MacOS X and Solaris. These OS have a high number of packages with errors that are introduced by the developers of the package itself when they release it. For the Windows and Linux operating systems, an important fraction of errors were unrelated to any change in the code of the package itself or any package dependency.

The majority of errors appear and are resolved within a few days without any developer intervention. The other errors requiring intervention and fixing by the package developers are primarily errors introduced when a package they rely upon was modified. Maintenance effort hence needs to be focused on fixing errors caused by others. This may become detrimental to package maintainability in the long run if the number of CRAN packages keeps on growing at the same pace.

While the current role of the automated error check for CRAN packages is to inform developers about whether their package conforms to the CRAN quality policy, we believe that R maintainers and developers could benefit from a more specific tool giving less general information than the current check and more information about the implications and problems raised by dependency changes.

ACKNOWLEDGMENTS

This research was carried out in the context of ARC research project AUWB-12/17-UMONS- 3. We thank Bram Adams for the numerous discussions we had on this research topic.

REFERENCES

[1] K. Hornik, "Are there too many R packages?" *Austrian Journal of Statistics*, vol. 41, no. 1, pp. 59–66, 2012.
[2] J. Ooms, "Possible directions for improving dependency versioning in R," *R Journal*, vol. 5, no. 1, pp. 197–206, Jun. 2013.
[3] K. Manikas and K. M. Hansen, "Software ecosystems: A systematic literature review," *J. Systems and Software*, 2012.
[4] S. Jansen, M. Cusumano, and S. Brinkkemper, Eds., *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013.
[5] M. Lungu, "Towards reverse engineering software ecosystems," in *Int'l Conf. Software Maintenance*, 2008, pp. 428–431.
[6] M. Goeminne and T. Mens, "Analyzing ecosystems for open source software developer communities," in *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, S. Jansen, M. Cusumano, and S. Brinkkemper, Eds. Edward Elgar, 2013.
[7] D. Messerschmitt and C. Szyperski, *Software ecosystem: Understanding and indispensable technology and industry*. MIT Press, 2003.
[8] D. M. Germán, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 243–252.
[9] J. Vouillon and R. Di Cosmo, "Broken sets in software repository evolution," in *Int'l Conf. Software Engineering*, 2013, pp. 412–421.
[10] P. Abate, R. D. Cosmo, R. Treinen, and S. Zacchiroli, "Dependency solving: A separate concern in component evolution management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012.
[11] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Why do software packages conflict?" in *Int'l Conf. Mining Software Repositories*, 2012, pp. 141–150.
[12] R. D. Cosmo, D. D. Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli, "Supporting software evolution in component-based FOSS systems," *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1144–1160, 2011.
[13] R. Di Cosmo and J. Boender, "Using strong conflicts to detect quality issues in component-based complex systems," in *Indian Software Engineering Conf.*, 2010, pp. 163–172.