A first look at an emerging model of community organizations for the long-term maintenance of ecosystems' packages

Théo Zimmermann

A first look at an emerging model of community organizations for the long-term maintenance of ecosystems' packages

Théo Zimmermann
theo@irif.fr
Inria, Université de Paris, IRIF, CNRS
Paris, France

ABSTRACT

One of the biggest strength of many modern programming languages is their rich open source package ecosystem. Indeed, modern language-specific package managers have made it much easier to share reusable code and depend on components written by someone else (often by total strangers). However, while they make programmers more productive, such practices create new health risks at the level of the ecosystem: when a heavily-used package stops being maintained, all the projects that depend on it are threatened. In this paper, I ask three questions. **RQ1**: How prevalent is this threat? In particular, how many depended-upon packages are maintained by a single person (who can drop out at any time)? I show that this is the case for a significant proportion of such packages. **RQ2**: How can project authors that depend on a package react to its maintainer becoming unavailable? I list a few options, and I focus in particular on the notion of *fork*. **RQ3**: How can the programmers of an ecosystem react collectively to such events, or prepare for them? I give a first look at an emerging model of community organizations for the long-term maintenance of packages, that appeared in several ecosystems.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Open source model**; **Software libraries and repositories**.

KEYWORDS

package ecosystem, maintenance, open source, fork, community

ACM Reference Format:
Théo Zimmermann. 2020. A first look at an emerging model of community organizations for the long-term maintenance of ecosystems' packages. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3392209

1 INTRODUCTION

Efficient software engineering and the conception of large software systems critically rely on reusable code [36]. Since the rise of open source ecosystems and package managers, software reuse has become a systematic coding practice, even for small and medium-sized software projects. Decan *et al.* [10] found a majority of packages depend on other packages in all seven ecosystems they studied.

Indeed, modern application-specific package managers have reduced the cost of sharing pieces of reusable code (libraries) that a programmer can extract from their project so that others can benefit from it. It has also made it much easier to rely on such a library in one's project. The consequence is that, nowadays, programmers will always look for a package solving their problem before considering tackling it by themselves. The extreme case is the one of trivial packages [1], which contain only one (sometimes very straightforward) function, and yet still get many users.

While creating and sharing a new package has now become very easy, does not cost much to the package author, and may even be beneficial to them, maintaining a package long-term can be a real burden [26]. Furthermore, authors may have little incentive to maintain some of their packages, considering that in some cases, they are not even using them anymore [11]. On the other hand, users that are currently using the library have much more incentive to contribute to maintaining it, but this is not always as easy for them as it is for the author.

The question of who should be responsible for maintaining an open source library is therefore far from trivial. And the answer may actually depend a lot on the way the ecosystem is structured.

This paper begins by asking a first research question (**RQ1**): how prevalent is the threat to ecosystems' health coming from packages that are used by multiple projects but could suddenly stop being maintained because a single person was responsible for them?

Next, we explore the various mitigation paths that projects can follow when faced with such a situation (**RQ2**).

The paper concludes with a last research question (**RQ3**): what can be done at the level of the ecosystem to mitigate such threats when they occur, or reduce their chances of occuring in the future? We will focus on a model of community organization, which emerged in several ecosystems, that can simplify the process of hard forking and provide a new home for unmaintained packages.

2 ESTIMATING THE PREVALENCE OF SINGLE-MAINTAINER PACKAGES

The main issue of small libraries is generally that they have fewer maintainers, most often a single maintainer, who might become less active or outright missing for a number of reasons. While the extreme case of trivial packages [1] can be trivially solved by copy-pasting the code and maintaining it within a larger source code base, I am interested in the frequent case of non-trivial libraries that are worth keeping as separate dependencies (to avoid duplicating innovation and maintenance work in the various projects depending on them) but are still small enough to have a single maintainer.

More specifically, I propose to focus on non-trivial libraries which are depended upon directly by at least two actively maintained software projects by distinct developers or developer teams: these libraries are thus worth maintaining separately. The goal of this section is to estimate the proportion of such libraries which have a single person maintaining them (i.e., a single developer has push-access, even if changes might be contributed by others through pull requests or e-mailed patches). Given that it can happen that maintainers of such libraries stop responding to requests and updating their library for extended periods of time, I claim they represent a very specific threat to the health of ecosystems.

This estimation will be based on the dataset provided by Libraries.io [28] (cf. the first companion Jupyter notebook [49]). This dataset includes a table of packages with source repositories, containing more than 3,000,000 entries. For this estimation, we focus on the 2,500,000 packages whose repository is located on GitHub, because the dataset contains more complete data for these.

A first filtering step of packages with more than one reverse dependency and a repository size of at least 10 KB (in an attempt to filter out trivial packages), makes this number go down to 257,000 packages, including 116,000 npm packages and between 10,000 and 30,000 PHP, Ruby, Python, and Maven packages.

To further filter only packages that are used in actively maintained projects from different owners (individual developers or teams), we extract the 800,000 GitHub repositories that were pushed to in the last six months (before the Libraries.io dataset was published) from a table which contains about 34,000,000 entries.

Finally, these data are joined by relying on Libraries.io's dependency table, which includes about 390,000,000 entries. This results in about 65,000 packages that are depended upon by two actively maintained projects from distinct owners.

Sometimes, many packages come from a single repository (more than 1,500 packages in the DefinitelyTyped repository[1] and 286 packages in the Babel repository —the next biggest monolithic repository). In this case, it is hard to estimate different maintenance indexes for the different packages, and giving the same index to thousands of packages would completely bias the results, so we keep only one package per repository (the most depended upon).

Out of the remaining 50,000 packages, **18% have just a single contributor** (which is worse than having a single maintainer).

For each package owner, we can query GitHub to know whether it is an organization and how many public members it has. About **33% of these 50,000 popular packages belong to organizations with at least two public members**. Belonging to an organization does not guarantee that the package will keep being maintained, but it should prevent against a maintainer disappearing without notice and no one having access to the repository.

For the rest of the package owners, I approximate the number of maintainers by querying for the number of assignable users. Assignable users are a super-set of collaborators with write-access:[2] they correspond to organization members with read-access (when the owner is an organization) and collaborators that were manually added to the repository itself [15]. **Of the remaining packages that were not part of an organization with two public members, only 33% have two or more collaborators.** This leaves a large proportion of packages at risk.

[1]DefinitelyTyped is a community repository gathering TypeScript type definitions for otherwise untyped Javascript packages: http://definitelytyped.org/.

[2]For privacy reasons, GitHub does not share the list of collaborators or people with write-access on a repository, but it does share the list of assignable users nonetheless.

2.1 Threats to validity

The goal of this section was to estimate the proportion of packages that are sufficiently popular to be used in two maintained projects by different owners, and yet have a single maintainer. Two kinds of errors could have affected the results: errors in the computation of popular packages and errors in the computation of single-maintainer packages.

Popular packages could have been missed, and in fact it is certain that this was the case, since some popular packages are hosted outside of GitHub. Another possible error could have been with the maintenance criterion. Some projects should be considered maintained even if they were not pushed to during a six-month period (some feature-complete packages may need less frequent updates [41], especially in a slowly evolving ecosystem). Furthermore, some projects may have been pushed to after Libraries.io last refreshed GitHub's meta-data about them, and thus the dataset may have contained outdated information that could have resulted in marking these projects as unmaintained. Therefore, the number of maintained packages depending on a given package could have been underestimated, and the package excluded because of this. However, the resulting sample of popular packages is still rather large (50,000 packages). Bias in this sample could result mostly from the exclusion of non-GitHub projects and of many packages hosted in monolithic repositories.

I computed dependencies between packages irrespective of package version numbers, so this could also lead to considering former dependencies that were dropped or replaced.

Then, the criterion that was used to approximate the number of maintainers is bound to have introduced some errors as well. However, I believe that it is more likely to have resulted in overestimating the number of maintainers of a package, rather than underestimating it. Indeed, it is not because someone can be assigned issues in a repository that they have access to all the required tools and credentials to be considered an actual maintainer of the package. Even if they are an actual maintainer, and are ready to take over from the previous maintainer for a while, if they do not have admin-access, they may not be able to nominate new maintainers, and this can harm the maintenance of the package after some time, in particular if they want to step down eventually.

Overall, I believe that these threats do not put much doubt on the main conclusion of my analysis which is that many popular packages have a single maintainer.

2.2 Related work

2.2.1 Project maintainers and developers. Yamashita *et al.* [46] analyzed the proportion of core developers in 2,496 GitHub projects. They used various criteria to define core developers, one of them using GitHub's collaborator API endpoint (through the GHTorrent dataset [18]). Unfortunately, access to this API has been restricted since then and GHTorrent does not include this data anymore. This is why I used the assignable users information instead.

My analysis is different from studies computing metrics such as *bus factor* (also called *truck factor*) [12, 39], and more generally estimating turnover risks [27, 34] within a software project, because I am focusing on the problem of who is able to integrate changes and publish new versions of a package. It can happen that popular packages receiving pull requests from many contributors still have a single maintainer that suddenly disappears. On the other hand, a project that is not subject to such risks, because it belongs to an active organization or has several maintainers, can still be subject to knowledge loss risks. Therefore, the two types of studies are complementary, and highlight different, but related, health risks that can affect a package ecosystem.

Avelino *et al.* [4] have found a significant proportion of projects having faced the event of a "truck-factor" developer stepping down, despite having only analyzed about 1,900 popular repositories. Less than half of the projects survived, generally because a new or pre-existing contributor took over. They interviewed these contributors that helped projects survive, and identified the difficulty of getting access to the repository as a significant barrier (when project maintainers had become unresponsive).

2.2.2 Unmaintained projects. Several studies have highlighted the fact that many open source projects are dormant or abandoned [22, 23]. However, a project threatening to become dormant does not pose the same risk to open source ecosystems depending on whether it has a lot of users, very few, or none beyond its author.

Valiev *et al.* [41] have found clear evidence that packages that have been able to gather a large community of users over time are much less likely to become dormant. This expected result does not contradict the observation that this risk is still present and strong for a number of popular packages.

2.2.3 Ecosystem health. Measuring ecosystem health is an important and active research question [21, 25]. My work relates to this literature by highlighting a health risk factor (single-maintainer projects) that could be integrated in health assessment frameworks.

3 COPING WITH UNMAINTAINED PACKAGES

When a package becomes unmaintained (the maintainer does not respond anymore to issues and pull requests and does not push new commits or versions), what can the projects using it do?

3.1 Removing or replacing the dependency

Upon discovering that a project depends on an unhealthy dependency, it can be time to reevaluate the usefulness of the latter. Sometimes, the functionality brought by the dependency is not that useful, or an alternative, healthier package could be used instead. Removing or replacing the dependency can, in such case, turn out to be beneficial, although the migration can bear significant costs to the project, not necessarily at the best of time.

Furthermore, this is a solution that each project has to evaluate on their own, and while it might be feasible for some projects to drop the dependency, it might be significantly harder for others. Having many projects migrate away from an unhealthy library can further reduce its chances of surviving, thus threaten other projects for which migrating is too costly.

Previous studies have mined data from projects having performed library migrations to suggest candidate libraries to migrate to [37] and to map between methods of two libraries [2, 38].

3.2 Vendoring

Vendoring a dependency is the process of copying its sources within the project's sources and building the whole thing, instead of installing the dependency with a package manager first and building the project by relying on the installed dependency.

Vendoring dependencies allows integrating patches proactively. Some of these patches might have been found in unmerged pull requests from external contributors. However, this solution cannot be a long-term solution because it is more work for everyone to have to integrate patches manually, and at some point new pull requests cannot continue to be based on an unchanged base branch.

3.3 Forking

3.3.1 Definition. Forking has a broad meaning today in open source.

Early academic works which studied forking considered only what is usually denoted today as *hard forks*. For instance, in their 2012 paper [35], Robles and González-Barahona give a definition of a fork that includes requirements such as having a new project name and a disjoint community. The Hacker's Dictionary [33] even specifies that the two code bases must be developed in parallel and have irreconcilable differences between them.

The "right to fork" is qualified by Weber [45, page 64] as an essential freedom of free software. Nyman and Lindman [29] claim that forking is the most important tool to guarantee sustainability in open source development, and that the right to fork has a major effect on governance, even in the absence of any forks.

With the rise of GitHub, forking has taken a new meaning. *Development forks* [13, Chapter 8] are copies of the sources where a contributor makes changes to the code in order to submit them for review through the pull request mechanism.

But even before GitHub, forking was much more common and much less definitive than hackers and researchers alike seemed to believe. Nyman and Mikkonen [30] observed the presence of many forks on SourceForge, including forks claimed to be temporary and hoping to get their changes integrated upstream (that can therefore be classified as development forks). They also noticed the phenomenon of forking a project because it seemed to be abandoned, and not because of some disagreement. While this should still be denoted as a hard fork, there is only one project under development after the fork, contrary to the definition of the Hacker's Dictionary [33]. This is the kind of forks that we are interested in, in this section. We denote this sub-type of hard forks as *friendly forks.*

3.3.2 Socio-technical issues when forking a package.

When to advertise a friendly fork? While it is easy for anyone to maintain a personal fork of a project, which contains the original code with some modifications on top, it may be difficult to decide when to advertise this project as a friendly fork intending to take over the place of the original, unmaintained project.

First, maintaining a fork of an unmaintained project for a long time without doing any advertisement is likely to result in duplicated work, as other persons interested in the project, but who are not aware of the fork, run in the same issues and prepare their own fixes. Zhou *et al.* [47] studied inefficiencies that may arise mainly due to lack of awareness of the work that was done in forks.

Besides, putting efforts into advertising a fork can pay back by bringing an influx of new contributions from developers that were interested in the project but were discouraged by the absence of feedback from the maintainer. Still, it requires time and commitment. The model of community forks that is discussed in Section 4 can help reduce the level of commitment required.

The time to wait until the source project is considered unmaintained can also vary depending on community expectations, and is rarely clear to anyone. While SourceForge's "Abandoned Project Takeover" page set a 90-day delay to get an answer,[3] CPAN's FAQ [19] informally sets a delay of one year without response before considering transferring maintenance of a module.

How to fork on GitHub? On GitHub, forking a project is as easy as clicking on a button. But, when preparing a hard fork, the new maintainer may wonder whether this is the right choice.

By default, forks on GitHub are not meant to take over a project: issues are disabled (but they can easily be switched on) and a prominent link to the source project is displayed under the project's name. Besides, code is not searchable in a fork unless it has more stars than its source [17] (which can take quite some time to get).

GitHub does not make discovering maintained forks very easy: the only way to learn about them is to display the fork tree, which is often very large. When the forks are too numerous, GitHub will not display the full list and the most important ones may be missing. The "Lovely forks" browser extension [40] helps developers discover notable forks by querying for them and showing them prominently, where GitHub would display a fork's source.

An alternative solution is to create a new repository manually, and to push the content of the original repository in it. It is also possible to contact GitHub staff to remove the fork status from an existing repository. They can also change the base directory in a fork network, but this requires consent from the original owner.[4]

Migrating issues and pull requests. GitHub does not provide any support for easily duplicating issues between two repositories. Doing so is nonetheless possible using a tool such as github2github [6]. Reusing the exact same numbers for imported issues is technically feasible. The advantage of doing this is that the code and the commit message frequently reference issues by their number, and importing preexisting issues ensures that these numbers continue to make sense. On the other hand, new maintainers might appreciate the ability to duplicate only a subset of issues they intend to solve.

If the fork was created using GitHub's fork button, it is also possible to manually recreate pull requests for every pull request that is still opened on the original repository.

How to publish updates to the package registry? Different registries and different ecosystems have different views regarding the transfer of a package to a new maintainer. Most support voluntary transfers, and some also support transfer to a new maintainer when the previous maintainer is completely unresponsive.

In some registries, all packages are scoped (the package name is prefixed by the name of the author), so unless an author explicitly gives access to a new maintainer, there is no way to continue using the package full name, and everyone will have to update their dependencies. On the other hand, it means the source project does not get a special status in the package index compared to its forks.

In some registries that support both non-scoped and scoped package names, keeping the same base name while adding a scope can be a way of marking the affiliation of the package to the original one.[5] This is also the technique that is used by the DefinitelyTyped repository (cf. Footnote 1) to publish type definitions corresponding to untyped Javascript packages.

In registries based on a shared repository of manifest files, it is technically easy to change the source of a package when publishing an update. The question of whether it gets accepted will depend on the registry's policy. For instance, MELPA's policy [24] says that forks will not be accepted except in "extreme circumstances".

Early archives where sources (and sometimes even bug trackers) are located on the platform make it technically even easier to change the maintainer of a package: CTAN, CPAN, CRAN and PEAR all have documentation regarding unmaintained / orphaned packages. CTAN specifies that modifications to a package should come from the package author or maintainer, new maintainers can be accredited by the current maintainer, but leaves the door open to discussing a solution when a package is unmaintained and the author is unresponsive [8]. CPAN administrators can transfer maintenance of a package to a new volunteer after sufficiently many steps have been taken to reach the previous maintainer and advertise the intent to take over the package [19]. CRAN has a formal orphaning process, after which volunteers can request to become the new maintainers [7]. For non-orphaned packages, transfer requires written agreement of the previous maintainer. PEAR packages can be marked as unmaintained, and may then be transferred to a new lead maintainer [32].

In general, there is a trust issue associated to allowing a change of maintainer for a package (as this means that someone can update a dependency to a new version without realizing the change of ownership), but this pertains to a much more general trust question in code reuse and package ecosystems. Because of this issue, npm is even considering restricting the rules for voluntary transfers [5].

We can see that forking, while often the best solution for the user community, puts a very large cost on the new maintainer when they do things right and try to organize the community around the new fork. And when forks are created but not properly advertised, it can only lead to duplication of effort. In the next section, I present a solution that alleviates the cost of forking.

4 ECOSYSTEM-LEVEL SOLUTIONS

4.1 Community forks to increase sustainability

While hard forks are a possible solution to the problem of unmaintained packages, we have seen that this solution puts a significant cost on the new maintainer. It can also put a significant cost on the user community when the package registry does not make it easy to switch the maintainer of a package, because a possibly large

[3]Cf. https://web.archive.org/web/20100609115535/http://sourceforge.net/apps/trac/sourceforge/wiki/Abandoned%20Project%20Takeovers

[4]As I was told in a private mail by a member of GitHub staff.

[5]As recommended in this Open Source Stack Exchange answer about npm: https://opensource.stackexchange.com/a/7025/5858

number of users will need to learn about the hard fork, evaluate if it is likely to be viable, and update their dependencies.

The transition period, which starts when people become aware that the previous maintainer has disappeared and only ends when the user community has massively adopted a hard fork, is a time during which it is likely that efforts are duplicated, potential contributions as pull requests or bug reports are wasted because they are being ignored or users stop submitting them, the package user base stops growing, or even shrinks, as people look for alternatives to migrate to, or even start their own from scratch, etc.

This cost can be deemed too high, especially if the hard fork itself also has a single maintainer, and thus risks suffering from the same issues a few years later. It is natural that the user community can anticipate this and will be reluctant to move massively to a hard fork that does not take steps to prevent this scenario to repeat.

A preventive measure would typically be the creation of a *community fork*. When a package raises sufficient interest and enough people are motivated to keep maintaining it together, they can host the new repository in a dedicated GitHub organization instead of a personal account and ensure that, at any time, there will be several administrators of this organization and several persons with credentials to publish a new version of the package.

However, most popular single-maintainer packages are likely to be too small for such a community fork to happen. To facilitate the creation of community forks, a possible model is to host them in a community organization dedicated to the long-term maintenance of important packages. This is the model that I present now.

4.2 A model of community organizations

In this model, a single informal organization is created (typically as an organizational account on GitHub) to host community forks in a specific ecosystem (typically around a programming language or framework). A place is dedicated to discussing organizational aspects of the community and to proposing new packages for inclusion (for instance the issue tracker of a meta-repository). The criteria for accepting a package may vary, but generally include having at least one person who volunteers to maintain the package.

Maintenance may actually be a community effort, but the advantage of having a designated maintainer for a package is to avoid diluting responsibility. However, volunteering to be the principal maintainer of a package is not a long-term commitment. The point of hosting packages within a community organization is that the maintenance responsibility can easily be transferred if a maintainer wants to step down or becomes unresponsive, as long as there are responsive organization owners and a new volunteer maintainer.

Such community organizations can also facilitate collaboration and encourage maintainers to share best practices. If all maintainers are given commit access to all projects, one can easily help with another package while its own maintainer is temporarily unavailable.

Hosting hard forks in such community organizations is likely to be a factor that will help users adopt them faster, as it guarantees against the risks associated with a new single-maintainer package.

Finally, the existence of such community organizations provides an exit strategy for authors of popular packages that would like to step down from maintaining them. They can submit their package for inclusion and transfer them to the community if accepted. Again, inclusion criteria may vary depending on the specific organization.

4.3 The case of elm-community

One early instance of this model is elm-community (https://github.com/elm-community), which was founded in November 2015. On July 5th, 2019, I interviewed Ryan Rempel, the founder of the organization, who explained to me how this organization came about.

The Elm package ecosystem had already got a culture of package forking and updating every time a new version of the Elm language was published and some original package authors failed to react. This was made easier by the fact that all package names are scoped in the Elm package registry. Consequently, the source package does not have a special status compared to its forks.

However, some impure Elm packages (containing what people used to call native code and now call kernel code [9], i.e., JavaScript) had to go through a formal "blessing" (whitelisting) process to be published in the Elm package registry. For such packages, forking and updating could not be done so casually, because it would additionally require going through this formal approval process. This specific issue was discussed on the Elm users' mailing list[6] after the author of a widely used package, containing native code, had been unresponsive for two months. During that time, a pull request with a trivial patch required to upgrade the package to the new version of the language (Elm 0.16) had been left unanswered.

Max Goldstein, an active community member who is now part of Elm's core team, suggested the creation of an "elm-community" GitHub organization "to steward the most important non-official packages". Ryan Rempel, another active community member and the author of the unmerged pull request, jumped on the idea, created the organization, forked the package, and submitted a whitelisting request for it, on the same day. Two days later, he created a meta-repository named "Manifesto" in which he described the purpose of the organization in a README and whose issue tracker served to host organizational discussions and package adoption requests.

Ryan told me that the reason he had reacted so quickly after the idea was first proposed was because he had viewed this as an opportunity to foster a new form of collaboration, that would be less disciplined and less centrally controlled than what was common in the Elm community. Indeed, he told me, the Elm community is unusually disciplined for an open source community, around a core team that has very specific ideas about what kind of participation is welcome. All his previous attempts to advocate a more open community had failed, and left him tired. So in this case, he started the community organization without really leaving any time to anyone to discuss the idea, invited five or six people from the beginning, and it turned out to be successful pretty quickly.

Shortly after, it gained some repositories that made more sense to develop collaboratively, e.g., a series of standard library extensions.[7]

The creation of a Manifesto repository was initially meant as a way of explaining the philosophy of the project, but also to allow issues to be used for organizational questions. The word "Manifesto" was slightly political, but the text of the README avoided any provocative content. According to Ryan, the text was rather abstract

[6]Cf. https://groups.google.com/forum/#!topic/elm-discuss/-GQJkWGdMvg/discussion
[7]Cf. https://groups.google.com/forum/#!topic/elm-discuss/wJPvZUql6v0/discussion

at the beginning, and others helped make it more concrete over time. The governance, in particular, remained voluntarily informal.

The idea to have a principal maintainer for each repository to avoid dilution of responsibility (which leads to issues and pull requests being left unanswered) was introduced about six months later by a member of the organization.[8] This change made explicit the rule that issues and pull requests are normally handled by the repository's principal maintainer, but in case of unresponsiveness, any other member can step in, and in case of long-term unresponsiveness, the maintainer is changed.

4.4 An emerging model

Elm-community is not the only, nor the first, instance of this model.

Vox Pupuli (https://voxpupuli.org) was founded in September 2014 to maintain Puppet modules (initially under the name of puppet-community) and currently hosts over 176 repositories and 139 collaborators. They have precise migration documentation [42] which clearly states a preference for repository transfer, but supports hard forks when package authors are completely unresponsive. They have also made efforts in recent years to promote their model so that other communities can inspire from it to create their own [14, 20].

Sous Chefs (https://sous-chefs.org/) was founded in May 2015[9] to maintain Chefs "cookbooks" (initially under the name of the Chef Brigade) and had a meta-repository from the start.[10] They also have a clearly documented forking and transfer policy [43, 44]. In particular, their forking policy states that hard forks are republished to the package registry under the original name with a "sc-" prefix.

Both organizations are pretty open to any repository transfer from members of the organization.

DLang-community (https://github.com/dlang-community), founded in December 2016, has stricter inclusion criteria than Vox Pupuli or Sous Chefs: packages are not transferred or created in the organization without it being discussed with other members, and the package being important to the community.

The following are direct or indirect elm-community descendants:

- ReasonML-community (https://github.com/reasonml-community) was founded in January 2017 under the name Buckletypes (inspired by the DefinitelyTyped repository, cf. Footnote 1). It was renamed in July 2017, and got a meta-repository influenced by elm-community in January 2018.[11] However, it has failed to get clear adoption guidelines, its meta-repository is almost unused, and it has recently taken almost a full year to host a fork of the very popular graphql_ppx package after its author had stopped responding.
- Coq-community (https://github.com/coq-community), founded in July 2018, was directly influenced by elm-community.
- OCaml-community (https://github.com/ocaml-community), founded in August 2018, was influenced by coq-community and elm-community. Similarly to DLang-community, it only accepts popular OCaml packages.
- React-native-community (https://github.com/react-native-community) was founded in July 2016, but got a "renaissance" period starting in December 2018 that was influenced by OCaml-community.[12]

Some organizations are similarly structured and intend to ease package maintenance after their maintainer has left or lost interest but do not explicitly support hard forks. Examples include the F# Community Incubation space (https://github.com/fsprojects/FsProjectsAdmin), Electron Userland (https://github.com/electron-userland), the Elytra group (https://github.com/elytra) which does not have a meta-repository or general guidelines, but does list the maintainers of each repository in their description, and advertise when a module is looking for a new maintainer. The Fluent Plugins Nursery (https://github.com/fluent-plugins-nursery) is explicitly intended for plugins that are not maintained by their original author but also states "we don't want to fork original authors' ".

Identifying community organizations. In the following, I present a process I used to discover such organizations (cf. the second companion Jupyter notebook [48]).

First, I listed 75 keywords that could be expected to appear in the name or the description of such organizations. This included keywords expressing collaborative work such as "collective", "maintain", "participate", but also keywords expressing what is being worked on such as "library", "module", "package".

I used GitHub's search, via the GraphQL API [16], to query for organizations which matched one of these keywords, and which had at least 5 repositories. GitHub's search only gives access to 1000 results, so when the number of results was above this limit, I further split the search using language filters. This first step yielded over 30,000 organizations (this is close to 15% of all GitHub organizations with at least 5 repositories).

The second step consisted in applying some filters on the results. Since I was only interested in community organizations, I filtered out the ones that had less than 10 public members and less than 10 assignable users on the most starred repository (as a way of estimating the number of collaborators for organizations with mostly private members). Since I was interested in organizations maintaining important packages, I also filtered out the ones whose most popular repository had less than 10 stars.

The third step was intended to further reduce the list to organizations that have received repository transfers. Unfortunately, GitHub does not give access to this information. I used a trick which consists in comparing the creation date of the organization to the creation date of its repositories. If an organization contains repositories that predate its creation, they have necessarily been transferred. Obviously, the converse is not true, so it could have resulted in underestimating the number of transferred repositories.

I manually browsed the resulting table of 938 organizations with at least two such transferred repositories. I used the name and description of the organization to infer its purpose. I eliminated many organizations whose description was something like "community packages for X" when the organization's website was actually the website of X. Indeed, this case is too frequent, and the reuse of

[8]Cf. https://github.com/elm-community/Manifesto/issues/16
[9]Cf. http://lists.opscode.com/sympa/arc/chef/2015-05/msg00091.html
[10]The first issue (https://github.com/sous-chefs/meta/issues/1) and the first commit (`efb426f`) were created three days after the initial mailing list announcement.
[11]See https://github.com/glennsl/reasonml-community-meta-proposal and https://github.com/reasonml-community/meta/issues/1.
[12]Cf. https://github.com/react-native-community/discussions-and-proposals/issues/63

the main product's website is a good indicator of the absence of a website or meta-repository specific to the community organization.

When an organization seemed to correspond to the type I was searching, I opened its website or GitHub page and looked for more information about it. It was frequent to find organizations with many repositories but no information on its principles and whether it accepted new members or new projects.

I am well aware that applying this series of filters is very likely to have resulted in missing out organizations that still fit the model I presented in Section 4.2. Nevertheless, the number of examples that I found, and the absence of relationship between many of them (in particular, between elm-community and the two organizations that predate it) leads me to think that **this model of organization is naturally emerging in open source package ecosystems** in answer to the recurring problem of single-maintainer libraries that I presented in Section 2. The fact that such organizations frequently got inspiration from one another shows that, while the need for such an organization is natural, the exact way of structuring it is not. Therefore, this contribution is important because, by surveying existing instances, we can bring useful information to practitioners wondering about the opportunity of founding such organizations, and how to structure them.

4.5 How does it compare to earlier models?

The idea of shared infrastructure to develop several projects together is not new. For instance, PEAR [31] was PHP's first package registry, but it was also much more. Packages had to go through a formal submission process to get accepted. Once they were, they got a repository and bug tracker. Finally, when a package was left unmaintained by its author, a new maintainer could be appointed [32].

However, there was an important difference in the motivations of authors applying to get their packages accepted into PEAR compared to authors, or interested users, proposing a package to a community organization today. As PEAR was PHP's only package manager and registry at the time (until support for alternative registries was introduced, starting in 2005[13]), authors submitted their packages to get it distributed, rather than for the shared maintenance model. For some maintainers, the model and infrastructure were convenient, but others preferred to host their projects elsewhere.

Another well-known example is the Apache Foundation, which hosts many open source projects and provides shared processes and infrastructure. The main difference with the model of community organization presented in this paper is that the Apache Foundation only accepts larger projects (all the projects have several maintainers and their own mailing lists [3]). The Apache Foundation model is therefore not suited to solve the issue of single-maintainer packages, and does not support hard forking (despite the Apache web server being itself a community fork of an unmaintained project).

4.6 Open issues, future work

I have presented a model of community organization for the collaborative, long-term maintenance of an ecosystem's packages, and I have identified numerous instances of this model. However, the method I used to find these examples is not exhaustive.

The use of GitHub's search to list possible candidates is good for exploratory work, but cannot be used beyond that: I have observed that the results obtained are very unstable when trying to fetch them again, and I have also found a number of bugs in GitHub's search filters. According to GitHub staff, this is because the search index is sometimes out of date.

Finding more examples will require coming up with more precise criteria to detect automatically this kind of organizations, probably starting from the complete list of about 2,000,000 GitHub organizations. Then, we could try to identify which characteristics of an ecosystem favor the emergence of such organizations.

Given that this is an emerging model, each instance is unique and it would be useful to come up with a number of parameters that can be used to describe them. Interviewing founders and participants would be helpful in that regard. Then, the next step would be to identify which of these parameters are associated with successful community organizations. For instance, it would seem that documenting the adoption process is helpful to ensure that people know what to do when a useful package in the ecosystem has been abandoned and would be a candidate for adoption, whereas the absence of such guidelines can lead to a lack of reactivity and duplication of effort with multiple people starting forks (as happened with the graphql_ppx library in the ReasonML ecosystem).

Finally, it seems important to systematically assess the impact of such community organizations on an ecosystem and, in particular, on the packages that get transferred or forked in the organization. For instance, Zhou *et al.* [47] studied inefficiencies in fork-based development, such as duplication of work or community fragmentation. We could try to evaluate whether the presence of a community organization helps reduce these inefficiencies, and under which conditions. This would provide concrete incentives to practitioners to create more instances of this model.

5 CONCLUSION

Modern and attractive package ecosystems are made of a multitude of small and large open source libraries. When a popular package is depended upon by many projects but has only a single maintainer, it creates a risk of the maintainer suddenly becoming unresponsive and the package not being updated anymore. I have shown that a large proportion of popular packages are single-maintainer packages and I have presented some mitigation paths that users of these packages can follow in such situations of unresponsiveness. Among them, the creation of a friendly fork is often the best for the community, but it can be costly to start a fork, and it does not help very much if the new fork is also a single-maintainer package. This is why users can create community organizations for the collaborative, long-term maintenance of an ecosystem's packages. Such organizations can reduce the cost of forking, while creating better guarantees for the future of the fork. I have shown that this model of community organizations has emerged in a number of package ecosystems and, as a first step toward a more systematic analysis, I have presented the case of elm-community in greater details.

[13]Cf. http://php-pear.1086190.n5.nabble.com/Questions-about-PEAR-amp-Composer-td36854.html.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? An empirical case study on npm. In *Proceedings of the 11th Joint Meeting on European Software Engineering Conference and Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 385–395.

[2] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Migration-Miner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level. In *Proceedings of the 35th International Conference on Software Maintenance and Evolution*. IEEE, Piscataway, NJ, USA, 414–417.

[3] Apache Software Foundation. [n. d.]. How it works. https://www.apache.org/foundation/how-it-works.html. Last accessed September 21st, 2019.

[4] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. 2019. On the abandonment and survival of open source projects: An empirical investigation. In *Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement*. IEEE, Piscataway, NJ, USA, 1–12.

[5] Adam Baldwin. 2019. The security risks of changing package owners. https://blog.npmjs.org/post/182828408610. Last accessed September 4th, 2019.

[6] Andriy Berestovsky. 2018. github2github. https://github.com/berestovskyy/bugzilla2github/. Last accessed January 22nd, 2020.

[7] CRAN repository maintainers. [n. d.]. CRAN Repository Policy. https://cran.r-project.org/web/packages/policies.html. Last accessed August 30th, 2019.

[8] CTAN team. [n. d.]. How can I take authorship of a package? https://ctan.org/help/become-author. Last accessed September 4th, 2019.

[9] Evan Czaplicki. 2018. "Native Code" in 0.19. https://discourse.elm-lang.org/t/native-code-in-0-19/826. Last accessed September 6th, 2019.

[10] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[11] Fernando Doglio. 2018. The latest npm breach... or is it? https://blog.logrocket.com/the-latest-npm-breach-or-is-it-a427617a4185/. Last accessed July 23rd, 2019.

[12] Mívian Ferreira, Thaís Mombach, Marco Tulio Valente, and Kecia Ferreira. 2019. Algorithms for estimating truck factors: a comparative study. *Software Quality Journal* 27, 4 (2019), 1583–1617.

[13] Karl Fogel. 2005. *Producing open source software: How to run a successful free software project*. O'Reilly Media, Inc., Sebastopol, CA, USA.

[14] Igor Galić. 2017. Building Puppet's unofficial forge community. https://opensource.com/article/17/6/vox-pupuli. Last accessed September 6th, 2019.

[15] GitHub. [n. d.]. Assigning issues and pull requests to other GitHub users. https://help.github.com/en/articles/assigning-issues-and-pull-requests-to-other-github-users. Last accessed September 3rd, 2019.

[16] GitHub. [n. d.]. GitHub GraphQL API v4, GitHub Developer Guide. https://developer.github.com/v4/. Last accessed June 26th, 2019.

[17] GitHub. [n. d.]. Searching in forks. https://help.github.com/en/articles/searching-in-forks. Last accessed September 3rd, 2019.

[18] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *Proceedings of the 9th Working Conference on Mining Software Repositories*. IEEE, Piscataway, NJ, USA, 12–21.

[19] Jarkko Hietaniemi, Elaine Ashton, Ask Bjørn Hansen, and Leo Lapworth. 1998–2011. CPAN Frequently Asked Questions. https://www.cpan.org/misc/cpan-faq.html. Last accessed on September 3rd, 2019.

[20] Dave Hollinger. 2017. Vox Pupuli — The Importance of Working Together. https://www.youtube.com/watch?v=28izTNK_-00. Talk at DevOpsDays KC, video last accessed September 6th, 2019.

[21] Slinger Jansen. 2014. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology* 56, 11 (2014), 1508–1519.

[22] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[23] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. 2013. Is it all lost? A study of inactive open source projects. In *IFIP International Conference on Open Source Systems*. Springer, Berlin/Heidelberg, Germany, 61–79.

[24] MELPA team. [n. d.]. Contributing a new recipe to MELPA. https://github.com/melpa/melpa/blob/master/CONTRIBUTING.org. Last accessed September 3rd, 2019.

[25] Tom Mens, Bram Adams, and Josianne Marsan. 2017. Towards an interdisciplinary, socio-technical analysis of software ecosystem health. arXiv preprint arXiv:1711.04532.

[26] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. 2019. Why do people give up flossing? A study of contributor disengagement in open source. In *IFIP International Conference on Open Source Systems*. Springer, Berlin/Heidelberg, Germany, 116–129.

[27] Mathieu Nassif and Martin P Robillard. 2017. Revisiting turnover-induced knowledge loss in software projects. In *Proceedings of the 33th International Conference on Software Maintenance and Evolution*. IEEE, Piscataway, NJ, USA, 261–272.

[28] Andrew Nesbitt and Benjamin Nickolls. 2017. Libraries.io Open Source Repository and Dependency Metadata. https://doi.org/10.5281/zenodo.808273

[29] Linus Nyman and Juho Lindman. 2013. Code forking, governance, and sustainability in open source software. *Technology Innovation Management Review* 3, 1 (2013), 7–12.

[30] Linus Nyman and Tommi Mikkonen. 2011. To fork or not to fork: Fork motivations in SourceForge projects. *International Journal of Open Source Software and Processes* 3, 3 (2011), 1–9.

[31] Pear group. 1999. PEAR: PHP Extension and Application Repository. https://pear.php.net/. Last accessed September 20th, 2019.

[32] PEAR Quality Assurance Team. [n. d.]. Taking over an unmaintained package. https://pear.php.net/manual/en/newmaint.takingover.php. Last accessed September 4th, 2019.

[33] Eric S Raymond and Guy L Steele. 1996. *The new hacker's dictionary*. MIT Press, Cambridge, MA, USA.

[34] Peter C Rigby, Yue Cai Zhu, Samuel M Donadelli, and Audris Mockus. 2016. Quantifying and mitigating turnover-induced knowledge loss: Case studies of Chrome and a project at Avaya. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, Piscataway, NJ, USA, 1006–1016.

[35] Gregorio Robles and Jesús M González-Barahona. 2012. A comprehensive study of software forks: Dates, reasons and outcomes. In *IFIP International Conference on Open Source Systems*. Springer, Berlin/Heidelberg, Germany, 1–14.

[36] Johannes Sametinger. 1997. *Software Engineering with Reusable Components*. Springer, Berlin/Heidelberg, Germany.

[37] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2012. Mining library migration graphs. In *Proceedings of the 19th Working Conference on Reverse Engineering*. IEEE, Piscataway, NJ, USA, 289–298.

[38] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2013. Automatic discovery of function mappings between similar libraries. In *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, Piscataway, NJ, USA, 192–201.

[39] Marco Torchiano, Filippo Ricca, and Alessandro Marchetto. 2011. Is my project's truck factor low? Theoretical and empirical considerations about the truck factor threshold. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*. ACM, New York, NY, USA, 12–18.

[40] Utkarsh Upadhyay. 2015–2018. Lovely forks. https://github.com/musically-ut/lovely-forks. Last accessed September 3rd, 2019.

[41] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 644–655.

[42] Vox Pupuli team. 2016–2018. Migrating a module to Vox Pupuli. https://voxpupuli.org/docs/migrate_module/. Last accessed September 6th, 2019.

[43] Dan Webb. 2017. Forking to Sous Chefs. https://sous-chefs.org/docs/forking/. Last accessed September 7th, 2019.

[44] Dan Webb. 2017. Transferring to Sous Chefs. https://sous-chefs.org/docs/transferring/. Last accessed September 7th, 2019.

[45] Steve Weber. 2004. *The success of open source*. Harvard University Press, Cambridge, MA, USA.

[46] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E Hassan, and Naoyasu Ubayashi. 2015. Revisiting the applicability of the Pareto principle to core development teams in open source software projects. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*. ACM, New York, NY, USA, 46–55.

[47] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2019. What the fork: a study of inefficient and efficient forking practices in social coding. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 350–361.

[48] Théo Zimmermann. 2019. Automatic discovery of community organizations for long-term package maintenance. https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Fcommunity-orgs%2FCommunity_organizations.ipynb. Last accessed October 13th, 2019.

[49] Théo Zimmermann. 2019. Evaluation of the prevalence of single-maintainer packages. https://mybinder.org/v2/gh/Zimmi48/thesis/master?filepath=notebooks%2Flibraries-io%2FLibraries.io.ipynb. Last accessed October 13th, 2019.