Challenges in the collaborative development of a complex mathematical software and its ecosystem

Théo Zimmermann
IRIF, Université Paris-Diderot
Paris, France
theo.zimmermann@irif.fr

OpenSym '18, August 22–24, 2018, Paris, France
ACM 978-1-4503-5936-8/18/08.
https://doi.org/10.1145/3233391.3233966

Abstract

This is a contribution to the OpenSym 2018 Doctoral Symposium. This paper describes my PhD objectives. As an insider in the Coq development team, I've worked at making the release process of the Coq proof assistant smoother and more automated, at opening the development to external contributions, and at shaping the ecosystem around Coq. I'm intending to evaluate how well-known software engineering techniques and results about open source software communities apply in the specific case of the proof assistant I'm studying.

Author Keywords

Open source software; release management; mathematical software; Coq; proof assistant.

Background

Coq is a *proof assistant* or *interactive theorem prover*, i.e. a software to write interactively mathematical proofs and to verify them automatically. It originated as a research project that started in 1984.

It regularly gained in popularity with important milestones like in 2004 when Georges Gonthier and Benjamin Werner completed their mechanized proof of the four color theorem using the Coq proof assistant. It was also used by many researchers to formally verify software systems. In 2013, its

initial developers were awarded the ACM Software System Award.

Coq is now one of the most used proof assistants. It is massively taught in undergraduate and graduate classes at many universities. It serves as the basis of large research projects involving dozens of researchers and receiving millions in funding.[1] It is also a tool for industrial verification projects.

This has some consequences for Coq. On the one hand, it is not just a research prototype anymore but a software on which the work of many people depends. It must therefore meet their expectations (in terms of compatibility between versions, improved performance, improved support for some usages) while continuing at the same time to be an object of study for researchers. On the other hand, it now has a large community of users which can be seen as a crowd of potential contributors.

The problem I am thus interested in is how the development of Coq can adapt to meet the new expectations while taking the most of the new opportunities arising from a more open development.

As an insider in the Coq development team since 2015, I have been a direct witness and important actor in the evolution towards a more open development, which encourages external contributions.

Goals

The evolution in the development process to guarantee more stability had started before I arrived with the nomination of a “release manager” and the objective to have time-based release cycles. I was, nevertheless, a witness of the first time-based release and a main actor of the second and third ones. My first contribution is my participation to this evolution. In my thesis, I will analyze the evolution and evaluate its consequences on the development of new features (was it hindered? did it impact the research around Coq?) and on the satisfaction of big users’ needs (has the system gained in stability and has this been useful to them?).

The real opening of the development of Coq (the proof language and the software itself by opposition to extensions such as libraries or plugins) to external contributions started with the first Coq Coding Sprint (later renamed Coq Implementors Workshop) to which I was a participant. In the following years, the development became more and more transparent and open to external contributions. My second contribution is my action in favor of such external contributions and a more open development team. In my thesis, I will analyze this evolution and evaluate how it helped get valuable contributions from new active contributors (how did it help get new contributors, who were they, and how useful were their contributions given the inherent complexity of the Coq software?).

The ecosystem around the Coq proof assistant (in which I include external packages such as libraries, plugins and editor support packages, external documentation in the form of tutorials, books, Q&A...) evolved as well, as a consequence of the growth of the user community and changes in the distribution of Coq packages that had started before I arrived. We passed from a centralized model to a more distributed model. In the centralized model, library and plugin developers were solicited to send their packages to the Coq development team who would then take care of maintaining them and distributing them further (in an archive of so-called “user-contribs”). In the new model, they are

[1] The DeepSpec project https://deepspec.org involves four top US universities, lists 78 collaborators and received $10 million in funding from the NSF.

self-maintaining their packages and encouraged to directly submit them to a package repository to reach users who are then able to install them using a specific package manager. My third contribution will be to the organization and shaping of this ecosystem by involving the community more and more, especially in the long-term maintenance of Coq packages and their diffusion. I launched a project for collaborative long-term maintenance of Coq packages called coq-community[2] inspired by the similar and already successful elm-community.[3]

A cross-cutting goal is to evaluate how well-known software engineering techniques and results about open source software communities apply in the specific case of the proof assistant I am studying.

Methods

To propose amendments to our development processes, I observe what issues arise with the previous way of doing. For instance, a lot of breaking changes were implemented and merged without being documented in the changelog, just because developers and reviewers forgot, and we got complaints from users. Then, most of the time, I propose and I test alternative ways based on standard practices in open source development. In the above example, there were several possible solutions (used in other projects): enforcing that pull requests always come with a changelog entry by adding an automatic check; automatically generating a changelog using information from GitHub pull requests and commit messages; or simply reminding this requirement to pull request authors using a pull request template with a few checkboxes. I implemented the latter because it was the lighter solution and it turned out to be pretty efficient (although exact measurements are still to be done). However, there is still a problem with some experimented developers stripping the template without reading it because they think it's not for them, and we will see in the future if there is a need to move to one of the other two solutions.

Furthermore, the open source way is in constant evolution itself so there is no definite best solution, and all open source development teams are experimenting a bit at the same time. It is quite useful to keep informed of what others are doing, and to be ready to reuse their tools (and to share ours). The GitHub platform itself, which we are using, is evolving quite fast these days, and we regularly experiment with new features to see how they fit in our practices and if they are of any help.

Sometimes, just reporting a problem in our development processes and having it discussed is enough for good ideas to emerge. This kind of meta-issues really took off after the move from our old bug tracker to GitHub issues (described in the results section). Documenting processes provides clear and immediate benefits such as helping new contributors gain in confidence or making it easier to swap some special roles like "release manager". But it can also be a good opportunity to discuss these processes. Indeed, more senior developers are often questioning the adequacy of the new processes and lack of proper communication around some changes sometimes created frustration or confusion.

I also intend to have repetitive tasks automated more systematically. I was particularly inspired in this aspect by the talk "Cyborg Teams: Training machines to be Open Source contributors" by Stef Walter at FOSDEM'18. Lots of such automations are natively proposed by the GitHub platform while some others can be easily added because they have been implemented and distributed by other de-

[2] https://github.com/coq-community
[3] http://elm-community.github.io/

velopers. However, we must be ready to adapt them or develop our own when what is proposed cannot fit in easily in our processes, would impose too radical changes or wouldn't help with respect to our objectives. For instance, I created a methodology to track pull request backporting using a GitHub project (a Kaban-like board with columns and cards). When GitHub added automation features to these projects, I started using them to manage this more efficiently, but because these features were quite limited, I ended up developing a bot that moves cards around for me and saves me a lot of time.

To involve the user community more, the best way is sometimes to directly call for contributions: on Coq-Club, the Coq users' mailing list, like I did once to get a volunteer to migrate the Coq FAQ, which was bitrotting, to the Coq wiki; or at the Coq workshop, like I intend to do to find participants to the coq-community collaborative maintenance project.

To evaluate the impact of changes in our processes, I would like to get objective measures as often as possible. Most of the time, I will collect these indicators from gross statistics obtained through GitHub APIs. Ideally, we want to compare them both to the same statistics before a change was implemented and to equivalent statistics over other open source projects when we can extract such statistics from previous research papers. For instance, there is an interesting analysis to be done on the predicting factors of the time it will take for a pull request to get merged, and a comparison to do with the general results obtained in "Wait for It: Determinants of pull request evaluation latency on GitHub" by Yu et al.[4]

When this is necessary, this should be augmented with qualitative evaluation by interviewing interested parties or by detailed exploration of specific examples.

Literature search and exploration of other important open source projects will also get a central place in my thesis to be able to present a comparative evaluation of our methods and to import well functioning methods from other projects.

Results

As mentioned in the background section, I have contributed to the new release process: I encouraged the use of milestones and labels (which took much more importance after the migration to GitHub issues, see below); and, more importantly, I was put in charge of the stable `v8.7` branch at a time when we switched from a model of preparing bug fixes for stable branches and merging them into the development branch to a model where almost all pull requests target the development branch and bug fixes and documentation updates are backported to the active stable branch. In this position, I participated in the beta and final releases of Coq version 8.7, and I managed the patch-level releases for this version. This gave me a good overview of the whole release process, which I formalized and documented.

Another evolution towards more stability was the switch to a pull-based development model (including for core contributors). This allowed for better code reviews and systematic automatic testing (*a.k.a* continuous integration). In a first phase of this switch, which spanned two releases and more than a year, a single person was in charge of merging all pull requests after ensuring they had been reviewed. This was too inefficient and too time-consuming for the person in charge, so we recently switched to a distributed merging process based on maintainership of components. I helped put in place this new process. While it had clear immediate benefits, we can start to observe patterns on the time it takes for a pull request to get merged depending on

[4] https://doi.org/10.1109/MSR.2015.42

the component it affects (and thus whose maintainer is in charge). It will be interesting to do a detailed data analysis of this switch and its consequences.

After observing the many drawbacks of our previous bug tracker, I managed to convince the rest of the development team to move to GitHub issues. One important aspect of the migration was to keep the existing bugs and their bug numbers as often as possible. I conducted the migration by adapting a script that had been designed for much smaller scale migrations and I used it successfully to migrate the 4900 existing bug reports to GitHub issues. Only 500 bug reports, whose numbers were taken by existing pull requests, had to be renumbered. I documented this experience in a blog post available at https://theoz.im/bugzilla and shared the migration script.[5] I think this migration had a positive impact both on the opening to new contributors (as most of them already had a GitHub account, especially the students and the contributors to other open source projects) and on existing developers. One of my arguments in favor of the migration was that communication was hindered by the heavy user interface of the previous bug tracker and it seems indeed that online communication among developers was more fluid afterwards. This would however require a real evaluation which I plan to conduct.

[5] My improvements are now being integrated upstream, thanks in particular to the involvement of a reader of my blog post who used my improved version of the script to perform a similar migration.