Forked and Integrated Variants in an Open-Source Firmware Project

Ștefan Stănciulescu
IT University of Copenhagen, Denmark
scas@itu.dk

Sandro Schulze
Technical University Braunschweig, Germany
sanschul@tu-braunschweig.de

Andrzej Wąsowski
IT University of Copenhagen, Denmark
wasowski@itu.dk

***Abstract*—Code cloning has been reported both on small (code fragments) and large (entire projects) scale. Cloning-in-the-large, or forking, is gaining ground as a reuse mechanism thanks to availability of better tools for maintaining forked project variants, hereunder distributed version control systems and interactive source management platforms such as Github.**

We study advantages and disadvantages of forking using the case of Marlin, an open source firmware for 3D printers. We find that many problems and advantages of cloning do translate to forking. Interestingly, the Marlin community uses both forking and integrated variability management (conditional compilation) to create variants and features. Thus, studying it increases our understanding of the choice between integrated and clone-based variant management. It also allows us to observe mechanisms governing source code maturation, in particular when, why and how feature implementations are migrated from forks to the main integrated platform. We believe that this understanding will ultimately help development of tools mixing clone-based and integrated variant management, combining the advantages of both.

I. Introduction

Code *cloning* [1], [2], [3] is the practice of creating new code by copying the existing code and modifying it to match a new context. Cloning is used *in-the-small* to reuse implementations of non-trivial algorithms and to reuse local program patterns such as 'boilerplate' code seen in many framework-based systems. Moreover, cloning is frequently used for experimental changes without putting the system's stability at risk [4], [5]. Some authors suggest that, for highly-specialized complex code, cloning might even be the preferred reuse method [6].

Cloning is also used *in-the-large* to create new system variants by *forking*. In such a scenario, an entire project is copied and the copy, a *fork*, is customized to meet new requirements. The practice of forking has been observed both in industry [7] and in open source projects [8], [9], [5]. Reuse of existing tested code decreases the time and cost of delivery [4] and raises confidence in product reliability [7]. Forking, so cloning in-the-large, is a remarkably easy and fast reuse technique, and it is the subject of this study.

Both cloning in-the-small and cloning in-the-large are part of the *clone-and-own* paradigm [2], [10], [11] that is recognized as a harmful practice, credited for decreasing code quality [4] and multiplying maintenance problems [12], [13], [4], [7]. A bug found in one clone can exist in other clones, thus, it needs to be fixed multiple times [14]. Even just locating all cloned code may be nontrivial. Unintentional parallel development of the same functionality in different forks increases implementation and test costs [15]. Finally, merging diverged code forks is very laborious.

Our first objective is to understand how far the known benefits and drawbacks of cloning in-the-small apply to forking. To this end, we investigate forking practices in Marlin, an open source 3D printer firmware project. Marlin is an appropriate study subject due to an unprecedented amount of forks created in a very short period. The Marlin project has been forked 1588 times in the period of 3 years and 3 months. We investigate the following research questions using Marlin:

- RQ1 What are the main reasons for creating forks?
- RQ2 How do ongoing project development and maintenance benefit from existence of many forks? To what extent do forks retrieve changes from their origins? To what extent do forks contribute changes to their origins?
- RQ3 To what extent known drawbacks of cloning in-the-small (e.g., difficulties in propagating changes) apply to forking? Are there any new challenges?

Historically, *forking* had a negative antisocial connotation. It denoted a community schism, when a project is split and an independent development starts in a diverging direction [16]. The term has acquired a less negative meaning since the arrival of distributed version control systems and, in particular, of Github, which introduced traceability and easy propagation of commits between forks. Github makes forking relatively easy. Consequently, forking has become a potentially viable way to maintain concurrent program variants. Forking *can* now be seen as a software reuse mechanism next to established examples such as object-oriented reuse patterns, aspect-oriented programming, or software product line architectures.

The relation of forking and product line architectures has been noted in several recent works [17], [10]. The central part of a product line architecture is an *integrated platform* gathering together the core assets. The integrated platform uses programming language mechanisms, for instance conditional compilation [18], to maintain multiple variants simultaneously. Recently, we have proposed a lightweight methodology that combines the integrated platform approach with forking, referred to as a *virtual platform* [10]. The virtual platform attempts to combine the flexibility and low initial investment

of forking with acknowledged benefits of software product lines (the integrated platform). It proposes to use advanced automatic traceability mechanisms to maintain concurrent development of clones, and to allow for an easy migration of mature software fragments to an integrated platform. This way, small changes as well as experimental code could created and integrated easily.

The Marlin project is of interest in this context as it uses both conditional compilation (in the role of the integrated platform) and intensive forking (for more ad hoc volatile changes). As such, Marlin and Github can be seen as a spartan prototype of the virtual platform idea. Our second objective is to understand how the mixed variant development works in Marlin and to derive detailed requirements for developing the actual virtual platform. We formulate the following research questions:

RQ4 Under what circumstances is forking preferred over integrated variability for creating and maintaining variants?

RQ5 What are the criteria to introduce variants using conditional compilation instead of forking?

RQ6 What are the criteria that lead to integrating a forked variant into the platform using conditional compilation?

Previous studies focused mostly on analyzing cloning in-the-small, and there is little empirical evidence on what benefits and drawbacks cloning in-the-large exhibits in practice. We provide insights into forking as a reuse practice that can be beneficial for researchers and tool developers. We analyze cloning in-the-large from the variant management perspective. We provide evidence as to when and why cloning in-the-large is used for creating variants, and when it is better to have explicit variability. Our study uses an open source ecosystem, therefore the study subject can be further reused by other researchers to develop tools and methods, and run empirical studies for their tools. We provide traceability links to source material for interesting cases of cloning, merging, concurrent development, and others.

In the remainder, we provide basic definitions in Sect. II. Then, Sect. III presents the subject system, our methodology and the experiment design. Section IV surveys the results and synthesizes key observations. We then discuss threats to validity in Sect. V, related work in Sect. VI, and conclude with Sect. VII.

II. Background and Definitions

A. Git and Github

Git is a distributed version control system that allows for local repositories, which can be set to point to a remote repository. In the local repository, the user can commit his changes. When needed, changes can be pushed to the remote repository via the *git push* command. Git employs a lightweight and simple branching system, with each branch being nothing more than a label attached to a commit.

Github is a hosting service for Git repositories, offering a platform for collaborative development. Github allows for copying repositories in a structured way. This mechanism, known as *forking*, creates a traceability link between the copied repository, *the fork*, and the original project. On Github, a user can create a pull request which resembles a traditional change request. A pull request can be created either in the same repository, e.g. to allow a team to discuss the change, or from a fork to the original project. It consists of a description, possible comments from users, and a set of commits.

B. Basic Definitions

Repository. A repository is a structured storage for a project. The content is organized by a version control system.
Commit. A commit is an atomic change that was applied to a repository. It uses a similar syntax to UNIX patches, with a message attached that describes the change.
Fork. A fork is a copy of a project created by cloning in-the-large. A *formal* fork has been made using Github's forking mechanism. An *informal* fork is a copy of an existing repository created simply by copying files elsewhere, without any automatic traceability links. An *active* fork is a fork that has either synchronized with the origin after its creation, or has had changes applied to it. An *inactive* fork exhibits no activity in the repository after the creation date.
Variant. A variant is a project that was cloned and modified to satisfy certain requirements. Variants can also be created by derivation from an integrated platform, given a configuration.
Pull request. A pull request is a change request that contains commits and information about the change. A pull request can exist in one of the following three states: *open*—when the pull request is created and awaits to be verified, *merged*—the pull request is accepted into the target repository, and *closed*—the pull request has been rejected.

III. Methodology and Research Design

A. Research Questions

In Sect. I we formulated six research questions aiming at two main objectives. First, we want to understand whether forking is useful for the community, and to what extent it bears the benefits but also drawbacks of cloning-in-the-small. Second, we want to investigate the relation between integrated and explicit variant management. In particular, we want to reveal when and why conditional compilation, forking, or a combination of both is used.

B. Subject System

Marlin is a 3D printer firmware that works with ATmega microcontrollers [19]. It has been created by reusing parts of two existing firmware projects, Sprinter and Grbl, to which new code was added. The firmware computes and controls the movements of the printer, by interpreting a sliced model. Marlin must be flexible enough to deal with different hardware and printer types. It has about 140 features, which can be controlled using compile-time parameters. At the time of our data retrieval (November 2014), the main Marlin repository contained more than 1500 commits, and it has been forked 1588 times. The high number of forks and the fact that Marlin has explicit variability, makes it a good choice for our study.

The project was initiated by one Github user, ErikZalm, in August 2011. It gained attention and popularity due to several improvements over Sprinter. Over time, more than 100

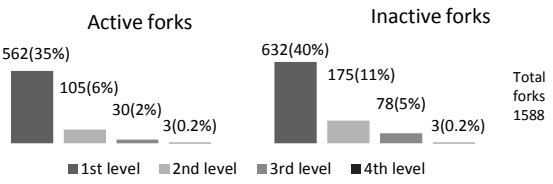

Fig. 1: Marlin's active and inactive forks, and forks' levels (percentages show the relative size in the set of all forks). The 1st level forks were created by forking main Marlin repository, the 2nd level forks were created by forking 1st level forks of Marlin, and similarly for 3rd and 4th level.

developers contributed to the project, out of which only 15 have direct commit access. Several other firmware projects are inspired by, or forked from, Marlin[1]. Besides hobbyists, 3D printer manufacturers use the Marlin firmware in their products.

Marlin is released under the GPL license. The project and its formal forks are hosted on Github[2]. In early 2015, the repository was transferred to a Github organization MarlinFirmware. In the paper we point to both, the new repository MarlinFirmware/Marlin and to the old one ErikZalm/Marlin (currently listed just as a fork of MarlinFirmware/Marlin).

C. Methodology

In our study we use mixed-methods combining qualitative and quantitative analysis. To obtain quantitative data, we built a simple tool that uses Github's public API to retrieve information about a repository, its branches, commits (and commits of each branch), issues, pull requests, forks and the owner of a repository, all as JSON files, which are then used to populate a database. The main purpose of this quantitative data is to get a comprehensive overview of the development process of Marlin and to identify points of interest that are further subject for detailed investigation.

To get insights why users fork and use conditional compilation, we classify the forks into two categories; purpose of the fork, and fork activity and nesting depth (Fig 1). We analyzed the corresponding commit messages (using key word search) of all forks using a simple heuristic in order to classify them by their main purpose (i.e., why a particular fork has been created). To obtain qualitative data, we analyze rejected pull requests to retrieve information about reasons for rejecting proposed changes from forks. In addition, we employ two short surveys directed towards active and inactive fork owners. The surveys contained both closed and open questions. They were available for ten days, after which we closed the possibility of receiving answers. We asked for the reasons to fork Marlin; what are the challenges encountered in getting their patches accepted; why they do not retrieve changes from Marlin and if they do it, how often they synchronize. We also asked about usage of conditional compilation. We distributed the survey invitation

[1]http://reprap.org/wiki/List_of_Firmware

[2]https://github.com/MarlinFirmware/Marlin

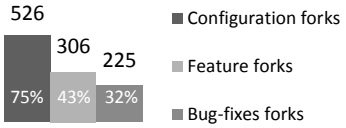

Fig. 2: Number of active forks used for configuration, developing features or bug-fixes (overlapping categories, percentages show the relative size in the set of active forks)

to 336 e-mail addresses (the other Github users did not add an e-mail address to their account), 185 belonging to owners of active forks and 151 to inactive ones. The response rate was 18.3% (34 respondents) for active fork owners, and 15.2% (23 respondents) for inactive forks. Finally, we interviewed two active maintainers of Marlin (in writing, open answers).

In the discussion, we link to exact commits by giving the Github user and the name of the repository, together with the first 8 characters of the commit hash. For example, ErikZalm/-Marlin #750f6c33, where ErikZalm is a Github user name, Marlin is the name of the repository belonging to that user and the commit hash is preceded by #. The commit hash is hyperlinked to the corresponding URL of the commit on Github, which can be conveniently followed if reading online. The above example points to https://github.com/ErikZalm/Marlin/commit/750f6c33e30ca16fab1ebe552a6b3422282bc66a. We also point to specific pull requests in the main Marlin repository, using the letter P followed by the number of the pull request, e.g. P1 points to https://github.com/MarlinFirmware/Marlin/pull/1.

Our database, survey questions, and other artifacts are available at http://bitbucket.org/modelsteam/2015-marlin.

IV. Results and Analysis

RQ1. What are the main reasons for creating forks?

We establish the purpose of forks using a term-based classification and substantiate the results with a quantitative questionnaire to fork owners. We first analyze the 700 active forks (Fig. 1) automatically. We establish the purpose of each fork analyzing commits on each branch heuristically, using the commit message. We divide the forks into the following three categories:

- *Configuration forks* are forks that change the configuration of the firmware. We detect them by checking if the main configuration files (files `configuration.h` and `configuration_adv.h`) are modified.
- *Development forks* are used to add new functionality. We detect them by matching the following search terms: *add, support, feature, new, added, implemented*.
- *Bug-fix forks* are used to fix defects. We identify them with these terms: *corrected, fix, bugfix, bug, fixed, replace*.

Commit messages of a single fork may match search terms of more than one category. In such case the fork is classified as belonging to all matching categories.

Figure 2 summarizes the results of the heuristic categorization. It indicates that 75% of the active forks have configured the

firmware. Detailed inspection shows that users of Marlin adjust the configuration to match their hardware, to enable/disable features or to fine tune parameters to ensure a good quality print. Some need to change the configuration to test a functionality that they are working on. Examples of configuration forks include: 3DPrintFIT #6343044c, jmil #04b8ef41, and Makers-Tool-Works #651b99d1. The configuration category is the most interesting. To the best of our knowledge, using version control is not a well-recognized way of handling variant configurations, while it is the dominating reason for forking in Marlin. The other two categories are expected and cover development of new functionality (43%) and bug-fixes (32%).

In order to verify the automatic, keyword-based classification, we manually analyzed a random sample of 40 active forks. For this analysis, we manually checked if changes have been done to the fork, and categorized the type of change into one of the three categories explained above. For each fork, we checked whether it has been classified correctly by comparing the manual and automatic analysis. We found that the precision of the heuristic on this sample is 97% for configuration forks, and 94% for both development and bug-fix forks, thus the data of Fig. 2 appears reliable.

Finally, we have approached the owners of the forks with a quantitative survey, asking them what were the reasons for creating their forks. Among the active fork owners, 62% (21 responses) report that they had originally intended to configure the firmware. This fraction is smaller than 75% of the actual number of configuration forks (Fig. 2). The intention to just configure the firmware was even more dominating among inactive fork owners (74% or 17 responses). The situation is the opposite for the non-configuration forks: 68% of active fork owners report that their intention was to contribute new functionality or modify the existing one. This is more than the actual number established heuristically: 54% of forks are used for new functionality or bug-fixes, which is the sum of the last two columns in Fig. 2, corrected for the intersection. Apparently, when you start with an intention to contribute new code, you cannot be certain to succeed. You may end up just configuring the code, even if you did not intend to do this.

Maintaining variant configurations in forks of entire projects is a very simple and effective mechanism. It does not require specialized configuration management or variability management tools. The fork owner has a reliable backup copy of the configuration, and the configuration can always be easily reconciled with upstream, if that becomes desirable. The Marlin community is extremely successful using this mechanism for the purpose. As mentioned above, some developers end up following this practice, even though this is not what they initially expected. It is actually somewhat surprising that this practice is not to be found in classic product line literature.

Observation 1. *Storing variant configuration data of a product family in forks of the entire project is a lightweight and effective configuration maintenance mechanism.*

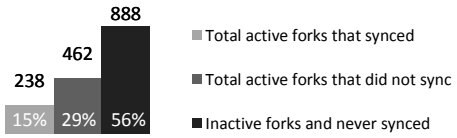

Fig. 3: Retrieval from origin (non-overlapping groups, percentages show the relative size in the set of all forks)

RQ2. How do ongoing project development and maintenance benefit from existence of many forks?

We answer this question by investigating to what extent there is a flow of contributions between the forks and main Marlin. We split the discussion into two subquestions and then synthesize.

To what extent do forks retrieve changes from their origins? To answer this question, we checked for each fork whether its branches contain any commits added to the main Marlin repository after the fork's creation date. We found that only 238 forks (15% of all forks) have synchronized at least once with the main Marlin repository (Fig. 3), which amounts to only 34% of active forks.

Observation 2. *Most forks do* not *retrieve new updates from the main Marlin repository.*

Marlin developers fork significantly more often than merge. This is a striking observation, given that merging of concurrent development strands is the key purpose of git. The forks in the Marlin ecosystem are characterized by a short maintenance lifetime (101 days on average). Once a fork achieves the desired functionality (the printer operates as expected) the incentives to maintain the fork decrease, and it becomes inactive (32% of active forks did not receive any commits between January 2014 and November 2014). Thus the period in which upstream changes are relevant for many developers is relatively short.

We confronted this hypothesis with the developers in our questionnaire. In the responses, only 18% of active fork owners synchronize monthly, and 6% synchronize weekly. Others do not synchronize at all, or synchronize irregularly. When asked, they state that the upstream changes are uninteresting for them, or that they do not wish to take in new changes as integrating them costs additional work. Merging new changes from upstream can be difficult and time consuming. One inactive fork owner explained that *I fear that my settings/calibration could change, sometimes I stay 1–3 months without changing the firmware of my printer*. This reinforces our understanding that most Marlin developers use Github to manage their variant, and not to collaborate with others.

At the same time, the *altruist* developers that want to contribute to the community synchronize more frequently. From 306 development forks, 142 have retrieved changes from the main Marlin repository. Moreover, 87% of pushed patches from development forks, come from those that synchronized. Being up-to-date with the main repository is key for producing clean up-to-date patches. This is consistent with our hypothesis that the maintenance span determines the need for synchronization.

To what extent do forks contribute changes to their origins? We found that only 202 forks (253 with forks that were deleted before our retrieval, and they are represented as unknown in pull requests by Github) contributed with patches to the main Marlin repository, so not even all feature development forks (306 in Fig. 2) have contributed pull requests upstream. Nevertheless, 714 commits have been integrated in main Marlin by merging pull requests. These 714 commits constitute 58% of all commits in the main Marlin repository, excluding the empty commits acknowledging merges. Most pull requests come from the first level forks; only seven come from the second level forks (Tbl. I). We conclude that Marlin is strongly supported and developed by the community.

An example of a contributed functionality is the *Auto Bed Leveling* (ABL) feature. A prototype of ABL was first implemented in a fork, which does not exist anymore at the time of writing. The commits have been accepted into another fork, akadamson/Marlin #728c355f (traced in our database which was populated before the original fork has been deleted). Then it was ported in alexborro/Marlin-BedAutoLev #0344dbfc to the latest version of Marlin and finally included in the main Marlin repository #253dfc4b. The feature was later improved in fsantini/solidoodle2-marlin #cc2925b7, and the improvements were accepted in the main Marlin repository #89a304fd. This example demonstrates how innovation and improvements happen thanks to collaboration of several developers that are distributed in space and time. Each of them has control over its own fork, which is the key for innovation.

Observation 3. *The ability to fork gives developers control over the code base, which encourages innovation. More than half of the commits in the main Marlin repository come from forks of Marlin.*

Let us now return to the question RQ2. The Marlin project was forked 1588 times in three years and three months (an average of 40 forks per month). As shown above, more than half of the commits in Marlin come from these forks.

Another benefit of easy forking for developers is showing up in testing and debugging. Testing 3D printer firmware is difficult, because maintainers do not have access to all the supported hardware. Hence, changes that are related to new hardware are usually tested by users having the corresponding hardware (e.g. P335, P572). During the life of the project, many users debugged problems, reported bugs, and contributed fixes developed in their own forks (e.g. P335, P594).

TABLE I: Contributions to the main Marlin repository. The last row refers to deleted forks whose level is unknown

Fork level	# forks	# pull requests (total)	# open pull requests	# merged pull requests	# closed pull requests
1	197	389	56	245	88
2	5	7	3	3	1
3 and 4	0	0	0	0	0
unknown	51	92	2	51	39

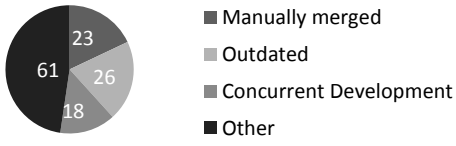

Fig. 4: Reasons for not merging pull requests. The numbers represent how many pull requests were rejected in each category. Other includes: closed by the pull request author (no reason), bad patch, pull request created on wrong repository, not fixing anything

Forking facilitates a gradual involvement of contributors. Fork owners gain experience working on their own forks. Once they gain reputation, they become committers in the main project. We have identified such cases both in our survey and in the interviews with the maintainers of the main project.

In summary, the development and maintenance of Marlin benefited from the multitude of forks in the following ways:

- Forks contribute new features and new hardware support.
- Fork owners test and improve the firmware on different hardware and configurations.
- Working on forks grooms new maintainers for the project.

RQ3. To what extent known drawbacks of cloning in-the-small (e.g., difficulties in propagating changes) apply to forking? Are there any new challenges?

We approach this question using the following methods: (i) studying the reasons for rejecting contributions, (ii) tracking how bug-fixes for important problems are propagated in the repositories and (iii) asking the fork owners about importance and challenges of receiving and contributing bug-fixes from upstream. All of these aspects can potentially reveal information about frictions in project management on the boundary of forks. We organize the discussion along the identified challenges starting with decentralization of information in forks, and moving to difficulties in propagating changes, redundant development, and other maintenance issues.

Decentralization of information. Decentralization of information is an issue that is specific to forking, where much more information is cloned than with cloning in-the-small. The modifications and extensions to Marlin are not kept in one neatly organized repository but in several hundreds of forks. For example, malx122/Marlin #69052359 added support for a second serial communication and malx122/Marlin #326c59f6 support for fast SD card transfer. These two features are not in the main Marlin repository, but they were actually taken from a fork (pipakin/Sprinter) of another firmware (kliment/Sprinter). Finding such features in the multitude of forks is extremely hard for the community members. This is consistent with an observation of Berger et al. [20] that an excessive use of clone-and-own to create variants in industrial projects leads to loss of overview of the available functionality. They name centralized information as a key advantage of integrated variability management and feature modeling over fork-based management. Also Duc et al. acknowledge that diverged code bases make it hard for

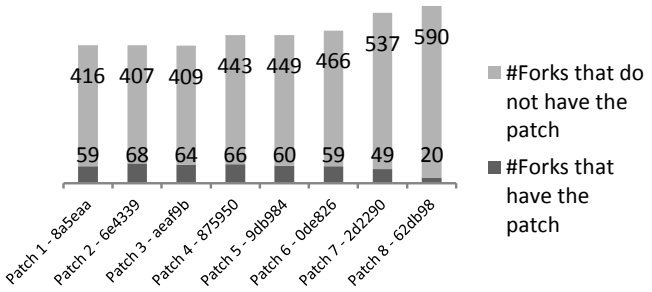

Fig. 5: Synchronization of active forks for patches. The sum of the two represents existing active forks at the time creation of that patch

individual teams to know *who is doing what*, and what features exist elsewhere [15]. This is a problem in Marlin as well, even though the individual forks do not depart far from the mainline. The sheer amount of forks makes it difficult to understand breadth of the available code.

Formal forking makes it easier to navigate the space of the available code compared to using entirely ad hoc forking. For example, the RichCatell/Marlin repository was not created using the forking mechanism of Github. This informal fork has several improvements including an updated auto-calibration feature, but it is far less known than the main Marlin repository.

Observation 4. *Decentralization of information in many forks is a challenge in fork-intensive development.*

Redundant development. The analysis of rejected pull requests showed that 18 pull requests (14% of all rejected pull requests) are rejected because of concurrent development (Fig. 4). The requested change either contained a feature or a bug-fix that already existed, or it was developed in parallel by two developers. This is a challenge not only for the contributors (e.g. P1087 or P223) but also for the maintainer who needs to have a good overview of all open pull requests to resolve the conflicts in the best possible way (e.g. P594). Berger et al. [20] confirm that concurrent development is a similar issue in industrial projects using clone-and-own.

Challenges in change propagation. The fact that forks do not retrieve changes from the main repository is problematic as fixes and new features are not propagated. In order to understand this phenomenon, we selected eight patches fixing important bugs in the main repository between January 2014 to November 2014, and verified if they were propagated to the forks. Figure 5 shows to what extent these patches have been adopted in forks. The light color part of the bar represents the number of active forks that have not pulled the patch, while the darker grey part represents active forks that have the patch (this only considers the active forks that existed before the patch was committed). For example, patch 1 in ErikZalm/Marlin #8a5eaa3c fixes a bug in a feature that may damage the printer. All the considered patch adoptions exhibit the same pattern.

Observation 5. *Propagation of bug-fixes is a problem for forking, just like for cloning in-the-small, even though git offers facilities for selective download of patches from upstream.*

At the same time, forks do not push changes back, so important fixes from the forks may never make it to the main repository. As many as 447 forks (63% of active forks) did not submit any patches to the origin. The survey data shows that one of the challenges is to prepare a robust pull request that does not break other features. A developer who works on his own fork, may find it difficult to take into account how his fix will affect configurations of all the other users. See for example a case of pull request P594 mentioned above, where a developer proposes to fix the problem for one hardware configuration, by removing the code that is necessary to make other configurations work. It is easier to maintain this general view on the variants, when integrated variability is used.

Other maintenance issues. When a project has many forks the maintenance becomes costly due to the large number of incoming change requests. An interviewed project maintainer explains that Marlin needs more developers and maintainers, as there are not enough people to support all the desired changes from the community.

RQ4. Under what circumstances is forking preferred over integrated variability for creating and maintaining variants?

In order to understand why developers prefer forking over integrated variability management, we investigated scenarios in Marlin's history that are typical of fork-based development. We selected the cases that appeared to be beneficial based on developer's statements or on our experience in variability management. This qualitative analysis included forks that are used to manage configurations and forks that develop features, but do not push changes upstream. We disregarded forks that push changes upstream as these have to integrate variation using conditional compilation. Since Marlin itself was created as a fork, we investigated its origin and the initial rationale.

Observation 6. *We have observed that Marlin developers preferred forking over integrated variability under the following reasons:*

- *S1. The fork extension has little relevance to other users.*
- *S2. The maintenance time span for the developed code is expected to be short.*
- *S3. The external developer has no control over the upstream project.*
- *S4. A developer wants to create experimental code.*
- *S5. An active project provides a good skeleton for adding new functionality.*
- *S6. A defunct project contains code that can be reused.*
- *S7. The developer wants to change the programming language.*

Next, we detail the above scenarios using concrete examples. Our data shows that there are 526 forks that did modifications to the configuration files. There are 316 forks (45% of active forks) that modified only the configuration files and made no other changes (S1). For instance, 33d/marlin-i3 #abaec3b3 configures the firmware to comply with a specific hardware. Some other forks mix configuration changes with other development changes 0xPIT/Marlin #70c7dde7, which makes it difficult to create pull requests containing just the new code and no

configuration noise. Both previous examples have not been updated afterwards (S2). Recall that 66% of active forks never synchronized with upstream after making their changes. Once the firmware is configured and running on the printer, new changes are not desired and no further maintenance is associated with these forks (S2). The average lifetime of a Marlin fork is 101 days, according to our data.

Forks commonly develop features for their own use, which may be highly experimental. For instance, martinxyz/Marlin #a8d59b1a modifies the IRQ functionality of the software and even adds an alternative IRQ code (#2a1c0766) for the stepper motor control in the firmware (non-standard IRQs are unlikely to be used). In this case, experimenting with code and adding new features that the original project lacks are the main reasons for forking, but the changes are, at least initially, not interesting to other users (S1, S4).

The fork jcrocholl/Marlin is a first level fork of main Marlin. It adds support for a new type of printer, a so called *delta*, that works differently than the normal Cartesian printers. Delta printers use spherical geometry and compute the location of the movements using trigonometric functions, such that the nozzle is not moved along the Cartesian planes. Before this extension, Marlin provided already support for some existing hardware and most of the needed software (S5), which made it easy to introduce the extension. From user's perspective this was a large qualitative leap, almost a new project though, as it supported hardware with completely different design. The main Marlin project was not affected in any way by these changes. Moreover, the developer had complete control (S3) over his fork allowing him to progress fast.

Originally, Marlin itself was created by cloning and extending parts of kliment/Sprinter and grbl/grbl. The Sprinter firmware was itself a fork of tonokip/Tonokip-Firmware, which was based on Hydra-MMM firmware. So heavy forking in this community predates Marlin's time. These earlier projects provided a good skeleton, from which Marlin could evolve into a solid standalone variant (S5, S6). There were few reasons to incorporate Marlin improvements into the upstream projects, as soon as the main developer realized that he has different goals and a different roadmap. The original projects would have never agreed to accept them anyways (S3, S4).

Another interesting example is the Traumflug/Teacup-Firmware firmware that started as a complete rewrite of the triffid/-FiveD_on_Arduino firmware. This fork decided to switch from C++ to C (S7). In such case integrating the differences between the two projects as variation points made little sense (S7). For Teacup, forking was the only possibility of development.

RQ5. What are the criteria to introduce variants using conditional compilation instead of forking?

Use of integrated platforms has been a subject to extensive studies in the field of product line engineering. The main text books of the field give criteria to select variation points and features [21], [22], [23]. Still, we ask this question in the context of Marlin, as the community uses both integrated and forked variants. The accumulated experience sheds light on the choice between the two mechanisms. We approach the question by qualitatively analyzing forks that developed changes involving preprocessor directives, excluding those that modify only the configuration files (configuration changes involve changing preprocessor directives). We supplement this data with answers to our survey and interviews.

Observation 7. *Marlin developers prefer integrated variability (conditional compilation) over forking in following situations:*

T1. The flexibility to use several variants is needed.
T2. Coding conventions expect usage of #ifdefs.
T3. Project maintainers require conditional compilation for new features submitted to main Marlin repository.

We identified 261 forks (37% of active forks) that introduced preprocessor directives in their commits. Conditional compilation is used in these forks for the same reasons as in other system level software [24]: to reduce use of memory, to disable functionality that is incompatible with current hardware, or to control inclusion of experimental code. Marlin supports printers based on 8-bit ATmega micro-controllers that have limited flash memory, usually 4–256kB. Seven out of 11 developers report use of conditional compilation for managing memory limitations (e.g., *Not all boards have enough space to run all the features so my feature was only compiled into larger chips*). In general, developers use conditional compilation to guard functionality that is optional, allowing it to be switched off either for themselves or for other users. Hence, flexibility of integrated variants (T1) is needed to meet memory requirements of different use cases and different hardware in the same fork.

Interestingly, many forks that do not contribute any changes to main Marlin use conditional compilation. Only 83 out of the 261 forks using this mechanism actually created pull requests (32%). They might have needed the flexibility for their own printers or experiments (T1). However, several survey respondents clearly state another reason—the coding conventions of the surrounding code: *That was how it was done by the guys before me, I didn't want to break the feel of the code.* (T2).

Finally, if a developer plans to contribute a feature to the main repository, he has to follow Marlin's coding guidelines. One of the main project maintainers states in the interview: *Every new feature contribution requires conditionals in some amount. New features that don't do this will be deferred until they do* (T3). Indeed, the pull request P594 discussed above, was initially rejected because it did not properly consider conditional compilation. Obviously, the Marlin maintainers enforce this rule, because the broad Marlin community needs the flexibility prescribed by criterion T1.

RQ6. What are the criteria that lead to integrating a forked variant into the platform using conditional compilation?

We define integrating a forked variant as either merging the entire fork or one of its features into the main platform. Both phenomena are of interest for researchers working on development modes mixing integrated and forked (virtual) variants [10], [25], [26], [17]. While several research groups are

attempting to build technical solutions for mixed development, we know little about how they should be used. The two previous research questions tell us under what conditions a particular mode can be selected. RQ6 asks about changing the integration mode for a variant. We answer it by analyzing history of variants integrated into the main Marlin. We learn that:

Observation 8. *Marlin fork owners consider integrating their forks into the main platform for the following reasons:*

U1. Integrating widely used variants that need to be kept in sync with upstream reduces effort and evolution cost.

U2. Integrated features are more visible and attract more users.

Observation 9. *A fork-based variant is integrated into the main Marlin platform under the following conditions:*

U3. The quality of the feature is within standards. It has been tested and is known to work as expected.

U4. Project maintainers accept to take over the maintenance, and the feature is aligned with project goals.

The *Auto Bed Leveling* feature (#253dfc4b), described above, has been created in a fork, later updated in another fork and finally integrated into the main repository. The integration took place after the feature has been tested and widely recognized[3] as well functioning (U3). On the other hand, if a feature is not working as expected and may introduce bugs or affect functionality, then it is not integrated. One such example is thinkyhead/Marlin #de725bd4, that adds support for SD card sorting functionality. This feature was not accepted in the main Marlin, because it causes problems in some specific cases (U4), but it was kept in the fork because it is demanded by users.

The fork for delta printers jcrocholl/Marlin has been integrated into the main Marlin repository (#c430906d, #6f4a6e53). Integrating the forked variant into the origin gave both visibility (U2) and lower maintenance efforts (U1). One of the maintainers stated that deltabot was merged *because it was clear and we knew it had been well-tested* (U3). Additionally, developers started to contribute changes to deltabot (P511,P568). In the deltabot fork, there were only nine pull requests created and only one got accepted. On the other hand, after deltabot was integrated into Marlin, there were created 20 pull requests related to Deltabot in ErikZalm/Marlin (55% more), out of which 14 were accepted (U2, U4). Interestingly, jcrocholl/Marlin remains a separate fork where the experimental development (S4) ahead of the main repository is happening. This allows the owner to continue development outside the control of Marlin project maintainers (S3).

V. Threats to Validity

Internal validity. We have classified commits and forks using keyword-based heuristics, and then employed this classification in a qualitative investigation of selected cases. The use of an imprecise heuristics could introduce noise. We did cross-check the precision of the heuristic on a small sample, obtaining a good precision. Thus, we believe that the reported numbers are representative. Also since the detailed analysis is qualitative, misclassification would have been detected in later phases.

Other quantitative data (contributions from forks, pull request classification, propagation of bug-fixes, etc.) have been obtained using exact algorithms and manual analysis. Care has been taken to eliminate human mistakes.

During our study the Marlin firmware continued to change. We have used an offline database obtained during a short time interval to minimize the risk of divergences in the data source. Then we used this database for quantitative queries.

It is possible that inactive fork owners synchronize directly with their local git repositories, without leaving traces in the remotes. Thus, it is hard to quantify how many actually synchronize or push changes outside of the pull request mechanism. Our survey verifies that this practice does not appear at a meaningful scale though.

External validity. Practices in other software domains might not follow those of Marlin developers. It is reassuring though that several observations are consistent with findings of independent studies of related problems. These were cited during our analysis and in the following section. Since little is known about mixed development of integrated and forked variants, any insight, even just into the firmware domain, is useful.

VI. Related Work

Cloning in-the-large. Several existing works investigate the reasons for forking [7], [15], [5], [27]. Developers who fork in industrial projects are reported to be motivated by the immediate availability of the method, freedom of control and independence from the old code base [7]. Easy access to validated code is also a frequently cited reason [7], [15], [6]. Both in open source software [27], [5] and in industry [15] forking is also motivated by project organizational matters (who has control over what code). For instance, in commercial projects, different teams have different roadmaps for their products, different risk management strategies, and different release schedules—all issues that can be circumvented by forking [15]. Open source developers may decide to fork in response to needs that are not recognized by the project's maintainers [5]. A group of developers can fork a project to start a new organization with different priorities and direction than the original one [5]. As we can see, both in closed and open projects forking centers around adding new functionality, which seems impossible in other ways [27], [5], [7], [15].

Forking comes at a cost, though. Maintenance becomes more difficult [7], [15], largely because of the lack of traceability between the forks [7]. Features and bug-fixes for different code bases are developed independently, thus duplicated code is created unintentionally [15]. A bug that occurs in one code base may exist in others [7], [14], and can remain unfixed [14]. Typically, the independent teams are unaware of changes that exist in other forks [15]. Few forks are merged back into origin and even fewer integrate code from similar or original projects [5]. Rubin et al. [28], [17] have extracted a set of fork

[3]Forum discussion: "Bed Auto Leveling.. check this out": http://forums.reprap.org/read.php?151,246132

management operators, based on three industrial cases. The operators support both managing forks and integrating them into a platform. They are meant to reduce the cost of forking.

Our study confirms the above findings in several ways. We have found several bug-fixes that are not propagated in other projects. Adding new functionality is also one of the main reasons for forking in our study, but often the intention is to share the changes. We also confirmed occurrences of redundant development. The present study differs in the following ways. First, our objective is to understand how forking functions as a reuse and variant management mechanism more holistically; not only the reasons for doing it, but also other aspects such as the extent of synchronization, or criteria for choosing forking vs integrated variability and the life cycle of variants that are eventually integrated. We observe that in Marlin many forks share their changes with the origin, unlike the prior study of Robles et al. [5] who state that very few forks are merged back and even fewer integrate code from the origin or similar forks. Second, our study subject is an open source system. It follows a typical collaborative community driven development process, without a central organization. This is very different than the industrial processes studied before [15], [7]. Perhaps more importantly, this means that all the data used in this study is available online. We have provided traceability links wherever possible. This way, researchers working on tools and methods can use our study as an index of entries to evaluation data for their results. Third, this study differs methodologically combining study of artifacts (commits, branches, forks, pull requests), interviewing developers, and surveying both active and passive users of the community.

Cloning in-the-small. Cloning in-the-small can also be beneficial [4]. For instance, developers of the Apache web server use subsystem cloning to support a large number of different platforms: 51% of code is cloned across the relevant subsystems [29]. A study of the projects hosted in the Squeaksource hosting service reveals that 14% of the methods are copied between projects [30]. Even maintenance is reported to benefit from cloning. Cloning can decrease maintenance risk for program logic, as it allows avoiding any impact on unrelated applications or modules [6]. Cloning allows to quickly implement a new functionality similar to an existing one. Cordy [6] reports that in some domains (financial software in his study) cloning is encouraged as it reduces the risk of introducing errors. Experienced programmers clone consciously with intention to reuse knowledge [31], [32]. Still, it is generally agreed that cloning in-the-small also introduces software evolution challenges. Roy and Cordy perform an in-depth analysis of clone detection, covering aspects from techniques and tooling, to advantages and disadvantages of cloning and taxonomies of clones [33]. Several questions remain open to this day, e.g., if code clones should be removed, encouraged or refactored.

Unlike all the above works, we are interested in cloning in-the-large (forking). While some drawbacks of cloning in-the-small apply to cloning in-the-large (e.g. fixing bugs in clones), there are some that only apply to forking (e.g. redundant development across projects). We provide a detailed analysis on the benefits and drawbacks of forking in an open source community. We took a high level perspective, asking process and architectural questions about variant management using forks and conditional compilation. These could not been asked in the context of cloning in-the-small, used to scaffold code rather than to create variants.

Github and its pull request development model. Only 14% of the active projects in Github use the pull request development model [34], but 79% of commercial users of Github report use of its fork and pull request workflow [35]. The most important factors that lead to acceptance of pull requests are code quality, code style alignment, and the technical suitability of the change according to a survey on 770 Github users [36]. Pull requests are often rejected due to poor quality of the code, failing tests, not adhering to project conventions styles, and newer pull requests that solve the same issue but have better quality.

We corroborate a lot of these results. For example, only 15% of Marlin's forks create pull requests. Marlin is not different than other open source projects in this respect. However, instead of analyzing the usability of Github, we seek understanding of fork-based and mixed variant management. The new knowledge is not in understanding how the pull request mechanism works. We study pull request to understand the dynamics and life-cycle of forked variants. For example, the existence of pull requests is just a proxy for generality of the variant's value proposition.

VII. Conclusion

We have investigated how forking and integrated variant management function in a lively open source community. Forks are used to manage variant configurations. Somewhat surprisingly, Github forks of Marlin are used more to create disconnected software variants, than to collaborate on software development. Most forks diverge from the mainline, possibly due to a short activity time. Forking contributes greatly to the creativity in the project, and more than half of the commits originate in forks. Still, forking poses several challenges to project participants: it leads to distribution of large amounts of information in an unorganized space, important bug-fixes are not propagated, and occasionally the same functionality is developed more than once. Finally, we have extracted criteria used by Marlin developers to decide whether a variant should be created as a fork or integrated into the main platform, and when the former should be migrated into the latter. To the best of our knowledge this is the first such list created empirically.

This study will be used to derive requirements for a tool that mixes development of individual forks and many integrated variants, and to propose a development method around it. We have also initiated a project to integrate several Marlin forks into a product line architecture, in order to create a documented and open product-line re-engineering case study.

Acknowledgments. We thank Danilo Beuche for pointing us to the Marlin project, and maintainers Bo Herrmannsen and Scott Lahteine for their valuable input on the Marlin project. This work was supported by ARTEMIS JU grant n°295397 and the Danish Agency for Science, Technology and Innovation.

References

[1] D. Faust and C. Verhoef, "Software Product Line Migration and Deployment," *Software Practice and Experience, John Wiley & Sons, Ltd*, vol. 33, pp. 933–955, 2003.

[2] T. Mende, R. Koschke, and F. Beckwermert, "An Evaluation of Code Similarity Identification for the Grow-and-Prune Model," *J. Softw. Maint. Evol.*, vol. 21, no. 2, pp. 143–169, Mar. 2009.

[3] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection," in *CSMR*, 2008.

[4] C. Kapser and M. W. Godfrey, "'Cloning Considered Harmful' Considered Harmful," in *13th Working Conference on Reverse Engineering*, 2006.

[5] G. Robles and J. M. González-Barahona, "A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes," in *International Conference on Open Source Systems: Long-Term Sustainability*, 2012.

[6] J. R. Cordy, "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 2003.

[7] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *CSMR*, 2013.

[8] N. A. Ernst, S. M. Easterbrook, and J. Mylopoulos, "Code forking in open-source software: a requirements perspective," *CoRR*, vol. abs/1004.2889, 2010.

[9] L. Nyman, T. Mikkonen, J. Lindman, and M. Fougère, "Perspectives on Code Forking and Sustainability in Open Source Software," in *International Conference on Open Source Systems: Long-Term Sustainability*, 2012.

[10] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Laemmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible Product Line Engineering with a Virtual Platform," in *ICSE*, 2014.

[11] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants," in *ICSME*, 2014.

[12] B. S. Baker, "On Finding Duplication and Near-duplication in Large Software Systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, 1995.

[13] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *ICSM*, 1999.

[14] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," in *Symposium on Security and Privacy, SP*, 2012.

[15] A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan, "Forking and Coordination in Multi-platform Development: A Case Study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014.

[16] E. S. Raymond, "Homesteading the Noosphere," in *The Cathedral and the Bazaar*, T. O'Reilly, Ed. O'Reilly & Associates, Inc., 1999.

[17] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," ser. SPLC, 2013.

[18] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[19] "megaAVR — 8 bit family of microcontrollers," http://www.atmel.com/products/microcontrollers/avr/megaAVR.aspx, accessed: 2015-07-01.

[20] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski, "Three Cases of Feature-Based Variability Modeling in Industry," in *MODELS*, 2014.

[21] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[22] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[23] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[24] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *Trans. Software Eng.*, vol. 39, no. 12, pp. 1611–1640, 2013.

[25] T. Schmorleiz and R. Lämmel, "Annotations as maintenance tasks in similarity management," 2015, submitted for publication. 11 pages.

[26] E. Walkingshaw and K. Ostermann, "Projectional Editing of Variational Software," in *GPCE*, 2014.

[27] T. Mikkonen and L. Nyman, "To Fork or Not to Fork: Fork Motivations in SourceForge Projects," *Int. J. Open Source Softw. Process.*, vol. 3, no. 3, pp. 1–9, Jul. 2011.

[28] J. Rubin and M. Chechik, "A Framework for Managing Cloned Product Variants," ser. ICSE, 2013.

[29] C. J. Kapser and M. W. Godfrey, "Supporting the Analysis of Clones in Software Systems: Research Articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 61–82, Mar. 2006.

[30] N. Schwarz, M. Lungu, and R. Robbes, "On How Often Code Is Cloned across Repositories," in *ICSE*, 2012.

[31] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in *Proceedings of the International Symposium on Empirical Software Engineering*, 2004.

[32] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, Sep. 2005.

[33] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Technical Report No. 2007-54, School of Computing, Queen's University, Kingston Canada*, vol. 115, 2007.

[34] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *ICSE*, 2014.

[35] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German, "Open Source-Style Collaborative Development Practices in Commercial Projects Using GitHub," in *ICSE*, 2015.

[36] G. Gousios, A. Zaidman, M.-A. Storeyy, and A. van Deursen, "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective," in *ICSE*, 2015.