Empir Software Eng
DOI 10.1007/s10664-013-9284-6

Management of community contributions

A case study on the Android and Linux software ecosystems

Nicolas Bettenburg · Ahmed E. Hassan · Bram Adams · Daniel M. German

Abstract In recent years, many companies have realized that collaboration with a thriving user or developer community is a major factor in creating innovative technology driven by market demand. As a result, businesses have sought ways to stimulate contributions from developers outside their corporate walls, and integrate external developers into their development process. To support software companies in this process, this paper presents an empirical study on the contribution management processes of two major, successful, open source software ecosystems. We contrast a for-profit (ANDROID) system having a hybrid contribution style, with a not-for-profit (LINUX kernel) system having an open contribution style. To guide our comparisons, we base our analysis on a conceptual model of contribution management that we derived from a total of seven major open-source software systems. A quantitative comparison based on data mined from the ANDROID code review system and the LINUX kernel code review mailing lists shows that both projects have significantly different contribution management styles, suited to their respective market goals, but

Communicated by: Per Runeson

N. Bettenburg (✉) · A. E. Hassan
Software and Analysis Lab (SAIL), Queen's University,
School of Computing Kingston, ON, Canada
e-mail: nicbet@cs.queensu.ca

A. E. Hassan
e-mail: ahmed@cs.queensu.ca

B. Adams
Département de Génie Informatique et Génie Logiciel,
École Polytechnique de Montréal, Montréal, QC, Canada
e-mail: bram.adams@polymtl.ca

D. M. German
Department of Computer Science,
University of Victoria, Victoria, BC, Canada
e-mail: dmg@uvic.ca

Published online: 05 November 2013

with individual advantages and disadvantages that are important for practitioners. Contribution management is a real-world problem that has received very little attention from the research community so far. Both studied systems (LINUX and ANDROID) employ different strategies and techniques for managing contributions, and both approaches are valuable examples for practitioners. Each approach has specific advantages and disadvantages that need to be carefully evaluated by practitioners when adopting a contribution management process in practice.

Keywords Software management · Software process · Measurement · Contribution management · Open source software · Best practices

1 Introduction

In recent years, open-source as a business model has gained in popularity and studies have documented benefits and successes of developing commercial software under an open-soure model (Krishnamurthy 2005). Companies like RedHat, IBM or Oracle, realize that collaboration with a thriving user and development community around a software product can increase market share and spawn innovative products (Hecker 1999). One of the major benefits of an open-source business model is user-driven innovation. As opposed to traditional, in-house development models, an open-source model gives the users of a software system the ability to actively participate in the development of the product, and contribute their own time and effort on aspects of the product that they find most important.

At the core of innovation are contributions, such as source code, bug fixes, feature requests, tutorials, artwork, or reviews, received from the community surrounding a software product. These contributions add to the software product in many ways, such as adding new functionality, fixing software defects, or completing and translating documentation.

However, involving a community in the development process may require significant changes to a company's structure and development processes, as both external and internal contributions need to be accommodated at the same time. This leads to a number of challenges that might not be obvious at first. For example, issues such as sharing of code between proprietary parts of the product and the open source parts, the need for sanitization of external contributions, and even legal aspects, such as usage of third party libraries, and export control, need to be taken into consideration by the contribution management process (Hecker 1999).

In addition, past research provides evidence that many open-sourced projects today struggle with contribution management, resulting in small communities and slow development progress (Capiluppi et al. 2003). To support practitioners in moving from traditional, proprietary development models towards an open-source business model, our work aims to distill best practices and empirical evidence on how successful, large open-source development projects do contribution management in practice. Our first research question is thus:

Q1: **How do successful open-source software projects manage community contributions?** Past literature provides only limited insight into how contribution management is carried out in practice. In the context of this research question,

we aim to learn about contribution management from documented processes and practices of 7 major, successful open-source projects, and abstract our observations into a step-by-step model that covers essential steps in the management process.
In a systematic approach, we derived a conceptual model of contribution management that spans five steps, from initial inception of a contribution to the final integration of the contribution into the product and delivery to the end-users.

In order to learn how successful projects do contribution management in practice, we selected two major open source ecosystems, the LINUX kernel (for the rest of the paper referenced to as LINUX), and ANDROID, that have put much time and effort into establishing their contribution management processes. Our case study aims at (1) evaluating their contribution management processes along three additional research questions (activity, size, time), and (2) to distill the individual strengths and weaknesses of two extreme contribution management processes to aid practitioners in establishing their own contribution management processes. In particular, the three research questions that we base our quantitative evaluation on are:

Q2: **To what extent are community members actively engaged in the development of the project?** We believe that the number and the activity of community members are a good proxy to measure the success of an open-source software project. Thus, good contribution management should encourage community engagement.
We find that both projects attract growing communities surrounding their software products. The more open and voluntarily managed contribution management of LINUX is especially successful in dealing with a substantial growth in submitted contributions.

Q3: **What is the size of received contributions?** We believe that the size of contributions is a valuable proxy for the effort (and also the importance and value) that went into the contribution. While small contributions do not necessarily need to be trivial (i.e., a small size contribution can still have tremendous business value), large and substantial contributions (such as new features, or complicated bug fixes) are more valuable for a project, than small and trivial contributions (such as fixing typos in the documentation). Previous research has shown that in practice contributors tend to submit smaller contributions, and peer reviews tend to favour smaller changes (Rigby et al. 2008). Good contribution management should not only encourage larger contributions, but also be able to handle larger contributions effectively.
We find that in contrast to previous research, both systems favour larger sized contributions. ANDROID's contribution management process includes tool support that helps reviewers cope with larger contributions and enables effective cherry-picking of smaller parts out of sizeable contributions.

Q4: **Are contributions managed in a timely fashion?** Good contribution management should be successful in dealing with contributions effectively with respect to time and effort to avoid contributors feeling ignored or becoming impatient.
We find that the timeframe in which external contributors are active in the project and thus available to respond to feedback on their contributions is limited. As a result timely processing is important to avoid losing valuable contributions.

We identify the main contributions of our work as:

1. A conceptual model of the contribution management process with the goals to: (1) methodologically derive an abstraction of the contribution management of successful, large open source projects from scattered project documentation, and (2) to provide a common basis for terminology and practices of contribution management.
2. Descriptive statistics and recommendations to practitioners based on case studies of two real world instances of contribution management processes and a quantitative assessment of their success.
3. An investigation of successful practices of contribution management with the goal to support practitioners who aim at establishing effective contribution management when moving towards an open source business model.

The rest of this paper is organized as follows. We present a conceptual model for contribution management that we derived from seven major for-profit and not-for-profit open source systems (Q1) in the first half of Section 2. In the second half of Section 2, we discuss howANDROID and LINUX realize contribution management in practice through a case study that follows our conceptual model of contribution management. In Section 3, we present our quantitative evaluation of contribution management in LINUX and ANDROID, which follows along the remaining three research questions (Q2–Q4). In Section 4, we discuss potential threats to the validity of our study and outline the precautions we have taken to balance these threats. We proceed with identifying and discussing related work in Section 5, followed by our concluding remarks in Section 6.

2 How Do Successful Open-Source Software Projects Manage Community Contributions (Q1)?

2.1 A Conceptual Model of Contribution Management

In order to establish a common ground for studying contribution management processes with respect to terminology and concepts, we first derive a conceptual model of contribution management. In the same way that architectural models create abstractions of actual instances of software implementations and give researchers a common ground for studies, our conceptual model aims at being a starting point for establishing common terminology when talking about contribution management.

The model of contribution management presented in this section was derived through a systematic study of publicly accessible records of processes and practices in this paper's subject systems, ANDROID and LINUX, as well as five additional popular for-profit and not-for-profit open source software systems from different domains. The seven software systems that we used to derive our conceptual model from are summarized in Table 1. These seven projects were selected among contemporary, prominent and important open-source projects, so we could understand how mature projects do contribution management. All projects have enough public exposure (press coverage, availability of documentation and discussion lists surrounding the development and contribution processes) to perform a systematic analysis of the

Table 1 Overview of for-profit (OPENSOLARIS, ECLIPSE, MySQL, ANDROID) and not-for-profit (FEDORA, APACHE, LINUX) open source projects studied to extract a conceptual model for contribution management

Project	Led by	Domain	Model	System size
OPENSOLARIS 111b	Company (Oracle)	OS environment	Commercial product features cherry-picked from open source code.	10M LOC
ECLIPSE 3.6	Foundation (Eclipse Foundation) funded through membership fees from individuals and companies	IDE and framework	Commercial product builds on top of open source product.	17M LOC
MySQL 5.5	Company (Oracle)	Database system	Support, certification and monthly updates for enterprises.	1.3M LOC
ANDROID 2.3	Company (Google)	Embedded device platform	Operates separately sold hardware.	73M LOC
APACHE 2	Foundation (Apache Software Foundation) funded through donations	Web server	Not-for-profit	2M LOC
LINUX kernel	Individual (Linus Torvalds)	OS kernel	Not-for-profit	14M LOC
FEDORA 14	Foundation (Fedora Project) sponsored by Company (Redhat), Community Governed	OS environment	Commercial product features cherry-picked from open source code.	204M LOC

surrounding documentation to derive a meaningful picture (Charmaz 2006; Strauss and Corbin 1990). Furthermore, we selected mature projects that are well-established in the open-source field. We derived system size through public source code statistics provided by Ohloh,[1] where available, and from press releases of the software systems otherwise.

We derive the conceptual model from publicly available documents about the seven subject systems, by following an approach known as "Grounded Theory" (Glaser and Strauss 1967; Strauss and Corbin 1990). Grounded Theory aims at enabling researchers to perform systematic collection and analysis of qualitative textual data with the goal to develop a well-grounded theory. The process starts with a maximally diverse collection of documents and follows three separate steps. The first step, called "Open Coding", consists of reading and re-reading all documents and identifying, naming and categorizing phenomena observed in the qualitative data. In the second step, called "Axial Coding", the analyst relates the categories derived in the first step to each other with the aim of fitting concepts into a basic frame of generic relationships. Through the third step, called "Selective Coding", the analyst distills the core concepts and abstractions of the observations in the qualitative data.

We start our derivation process on contribution management practices in Open Source Software (OSS) by first collecting and analyzing the publicly available documentation for each project, press releases, white papers, community mailing lists, and websites that document each project, as well as research literature in the area of open source software engineering. We started our abstraction of a common model by understanding the workflow that a contribution undergoes in each project. Some projects, such as ANDROID provide very detailed documentation,[2] whereas workflow in other projects is documented less explicitly by the members of the project themselves and needs to be recovered from more anecdotical sources. For example, in OPENSOLARIS, we recovered workflow information from multiple sources, in particular, the community Wiki and development mailing lists. Following the example of previous research in the area (Asundi and Jayant 2007; Rigby et al. 2008), we then looked for commonalities across all projects and finally divided the contribution management processes into individual steps that are common across all projects.

An inherent threat to the validity of such a derivative process is that we can claim neither completeness, nor absolute correctness of the derived theory. However, our aim was to derive a first abstraction, which we leveraged from the records and descriptions of actual implementations of contribution management processes in practice. This abstraction serves on the one hand as a starting point for discussing contribution management throughout our study on a scientific basis, and on the other hand we hope that future research will pick up and incrementally refine this abstraction with what is known in the field, similar to conceptual models of software architecture.

Overall, the derived conceptual model consists of five phases that a contribution undergoes before it can be integrated into the next release of a software product and

[1] http://www.ohloh.net

[2] http://source.android.com/source/life-of-a-patch.html

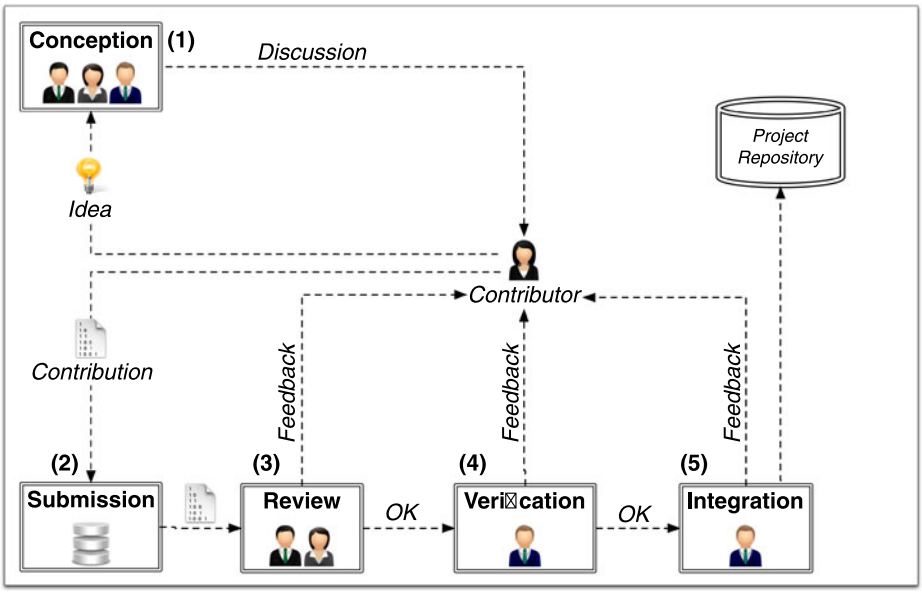


Fig. 1 Conceptual model of the collaboration management process

be delivered to the community. In the following subsections, we discuss each phase in the order of labels presented in Fig. 1, and illustrate each phase with concrete examples from the software systems that were analyzed.

2.1.1 Phase 1: Conception

Similar to classical software development, prospective contributors with an idea for a new feature or bug fix often seek early feedback and support from the rest of the community, as well as the internal development teams, to work out their ideas into concrete designs. Such discussions usually take place in public project mailing lists (e.g., OPENSOLARIS, APACHE), issue tracking systems (e.g., ECLIPSE), Wikis (FEDORA), and/or special purpose discussion forums (MySQL).

The outcome of the conception phase is either a concrete design (usually after multiple rounds of feedback), or a rejection of the proposed contribution, if the idea does not align with the project's or the community's goals. The conception phase is not mandatory—in some projects contributors skip this phase altogether and start with a concrete source code submission that was designed individually.

2.1.2 Phase 2: Submission

Once the design for a contribution has been fleshed out in source code, a contributor submits the contribution through the submission channels provided by the project. Since many of these submission come from external contributors (community members), intellectual property infringements are a substantial concern (German and Hassan 2009). All seven projects that we studied acknowledge this risk and have established policies for their submission processes that guarantee traceability of the submission to the original author.

For example, ECLIPSE, FEDORA, MySQL and APACHE completely disallow contributions through mailing lists, as the identity of the sender can not be verified. Instead, they require a submission to be carried out formally by opening a new record in their issue tracking systems.

2.1.3 Phase 3: Contribution Review

After a submission has been submitted for consideration, it will ideally reach senior members of the project (even though there is no guarantee that this is always the case, as our data on the LINUX system demonstrates). All seven projects require a formal peer review to be carried out for every submitted contribution. Contribution review has the following three goals.

1. *Assure Quality*. Senior developers may catch early on obvious issues of the contribution and possible source code errors, and give the contributor a chance to address these problems.
2. *Determine Fit*. As community members are often unaware of internal development guidelines and policies, the primary goal of the review phase is to determine the overall fit of the contribution for the project and ensure that contributions meet the established quality standards.
3. *Sanitize Code*. Reviewers check contributions for programming guidelines and standards, inappropriate language, or revise comments intended for internal viewing only. As part of the sanitization process, developers may also review the contribution for use of third-party technology, such as usage of external libraries whose licensing might not align with the project (German and Hassan 2009), as for example practiced in the ECLIPSE project.

The review phase has three potential outcomes: a contribution is accepted as-is, a contribution needs to be reworked, or a contribution is rejected. In case there are concerns with the contribution, reviewers can give feedback to the contributor, who is then expected to either address any concerns raised, or abandon his contribution.

2.1.4 Phase 4: Verification

After a contribution passes the review phase (often after multiple iterations), senior members of the project team or a project lead need to verify whether a contribution is complete and free of software errors (e.g., making sure the contribution passes all regression tests). The verifier is the person who has the final say on whether a contribution gets accepted or not. If any problems arise during the verification phase, the verifier(s) can give feedback to the original contributor, who can then resubmit an updated revision of the contribution (back to Step 2).

Common reasons for contributions being rejected during the verification phase include software errors, incompatibilities with the most recent version of the project repository (e.g., they target an out-dated branch of the software that is no longer actively developed or maintained), or strategic decisions (Wnuk et al. 2009). The verifier has the ultimate say and can reject contributions that received positive reviews in the previous phase if he does not see a fit for the contribution in the long term direction of the project. For example, the contribution correctly implements a certain feature, yet an alternate version for the same feature is already planned to

be copied from another upstream project that also implemented the same feature independently.

Since verification is a tedious step, some projects try to automate or outsource this process. For example, in FEDORA and ECLIPSE testing during the verification phase is crowd-sourced through nightly builds (daily updated builds that are not meant for public release), which contain the latest contributions for testing by the community. In addition, build and testing infrastructures and tools such as HUDSON[3] or JENKINS[4] are becoming increasingly advanced and enable (semi-)automated verification of contributions in the context of the existing software.

2.1.5 Phase 5: Integration and Delivery

If a contribution has passed the peer review, is technically sound, and has been verified, it enters the integration phase. The goal of this phase is to integrate a contribution, often together with all other contributions that have passed review and verification, into the software product, and to ultimately deliver it to the community. Integration of a contribution is often challenging, as the contributed code may stand in conflict with the source code of other contributions, as well as internal changes. If integration fails, contributors are often required to adapt their contributions to remove conflicts and work together with the most recent revision of the development repository.

Strategies for integration range from an immediate merge into the publicly available source code repositories, to delivery of contributions as part of a release schedule, such as daily builds, or official releases (Duvall et al. 2007). For special contributions, such as critical bug fixes, or high-impact security issues, strategies for a fast-tracked integration of contributions are valuable for reducing ill-effects.

2.1.6 Intellectual Property (IP) Management

Interestingly, across all projects that we studied, we found only a single instance of a particular IP management process—in the case of ECLIPSE, a closed-access tool called IPZILLA[5] is used to internally check for IP issues with contributions from third parties. However, we found a number of "best-practices" documented across different projects that support the management of Intellectual Property through keeping track of contributor's identities by the following means:

1. **Login Credentials.** For ECLIPSE, FEDORA, MySQL and ANDROID, users need a valid and active registration in the online system used to facilitate the contribution process (BUGZILLA, GERRIT). Users are required to provide their name, email address (and in some cases a postal address) to successfully register in the system. Contributions are then associated to their unique user id or user handle in the system. During the registration mechanism of the GERRIT tool in ANDROID, a user has to explicitly agree to, and sign a CLA (Contributor License Agreement) with Google that covers the transfer of IP rights for any submitted contribution.

[3] http://hudson-ci.org/

[4] http://jenkins-ci.org/

[5] https://dev.eclipse.org/ipzilla/

2. **Formal Registration.** For APACHE[6] and OPENSOLARIS,[7] users are required to print a special form called the Contributor License Agreement (CLA), sign it and mail, email, or fax it to a central authority to become registered as a code contributor. Only contributions from successfully registered individuals are considered.
3. **Developer Certificate of Origin.** For LINUX, contributions are submitted by electronic mail. Instead of contributor submitting an individual contributor licensing agreement (CLA) such as we have seen in 1. and 2., emails are required to contain a "Sign-Off" field identifying the contributor through a name and email address pair. The version control system (GIT) tool includes a special command line option "–signoff" to automatically sign submissions to the code repository with the credentials that a user has provided in the tool configuration.
4. **Firewalling.** A common practice across all seven projects is to further restrict the rights for modification of the main source code repository to a small set of "committers". Individuals earn commit rights by demonstrating technical skill through continued contribution to the project as non-committers (in this case another committer needs to sponsor their contributions), and by gaining social reputation, i.e., the more well-known and trusted individuals are by their peers, the more likely they will be given permissions to change the code directly (Bird et al. 2007b).
5. **Internal Review.** Through an interview with ECLIPSE developers we learned that the ECLIPSE Foundation carries out internal reviews of submission to check for third-party license conformity. We could not find any such practice documented for any of the other six projects, but suspect that similar practices are in place even though they are not explicitly described.

2.1.7 Relationship to Other Socio-Technical Participation Models in OSS

Since the beginning of the Open-Source Software development phenomenon, researchers have sought to understand how and why developers join an open-source project, and what their properties of participation in the project's development process are.

For example, the *private-collective* model presented by Hippel and Krogh (2003) conjectured how external users contribute to software projects, in order to solve their own problems on the one hand, and technical problems that are shared among the community surrounding the software on the other hand. In their study of major open source projects, von Hippel and von Krogh find that such external contributors freely reveal their intellectual property without commercial interests, such as private returns from selling the software.

In particular, for the APACHE webserver, and the FETCHMAIL email utility, von Hippel and von Krogh describe how the private-collective paradigm of providing contributions to a software for a public good impacts the organization and governance of traditional (commercial) development projects (Hippel and Krogh 2003). In particular, von Hippel and von Krogh conjecture the existence of governing entities

[6] http://www.apache.org/licenses/icla.txt The Apache Software Foundation Individual Contributor License Agreement (CLA).

[7] http://www.opensolaris.org/os/sun_contributor_agreement The Sun Contributor Agreement (SCA).

(team leaders, core-developers) and project resources, which our study confirms and describes in detail in the next section. For example, we observe in both of our case study subjects the existence of leadership roles as proposed in the private-collective model. However, the setup of these leadership roles differs significantly across both our case study subjects: while the LINUX project is governed in a hierarchical, pyramid-like fashion of increasing level of authority, with a "benevolent dictator" at the top and a hierarchy of meritocratically chosen lieutenants below, we find that the ANDROID project is governed by a two-tier "board of directors" approach, where the authority is in the hand of Google employees.

While the private-collective model studies open-source participation and contribution from a user incentive-level, the participation model presented by Sethanandha et al. focuses on understanding how individual pieces of code in the form of patches are contributed by external users and are handled by the project (Sethanandha et al. 2010). In contrast to the work by Sethandha et al., the conceptual model presented in this paper covers a more complete picture of the contribution process, beyond the submission and handling of patches. In particular, our conceptual model integrates conception, verification and integration phases, which are crucial parts of the overall management of contributions from users.

2.2 The Contribution Management Model in Practice - A Case Study on ANDROID and LINUX

To investigate how successful projects perform contribution management in practice, we picked the LINUX kernel and ANDROID as examples to learn from. This choice is motivated in particular by (a) LINUX being a prime example of a thriving, long-lived project, which spearheaded the open source movement and (b) ANDROID being a for-profit open source project, which is backed by a very successful major software company (Google). Even though ANDROID is publicly available for free, the underlying business interest is still for-profit, as the mobile platform allows for sale of search, advertisements, and mobile apps, and hence the product is treated like most commercial software. The selection of both projects was based on four key characteristics:

1. Both projects are especially successful in the open source software domain.
2. Both projects receive a large quantity of contributions from their communities. These contributions help evolve the product, perform corrective and perfective maintenance, as well as a multitude of extra services (e.g., creation of graphics, art, tutorials, or translations), and extend the product halo.
3. Both projects are system software, so we can observe how they compare.
4. One project is for-profit, the other not-for-profit, which might effect the contribution management processes.

2.2.1 Data Collection

For both projects, we carried out a quantitative analysis by analyzing the available source code repositories, mailing lists discussion repositories, and a qualitative analysis through inspection of publicly available web documents, and project documentation. For LINUX, we also took experience reports and previous research (Mockus et al. 2000; Rigby et al. 2008) into account. For the ANDROID system, we

relied on Google's publicly-available documentation of processes and practices, as well as empirical knowledge derived from analysis of the source code and available data from the GERRIT code review system. We describe below our data collection steps in more detail.

LINUX LINUX contributions are submitted through the LINUX kernel mailing lists. There is one global mailing list, and multiple specialized mailing lists for the different kernel subsystems. To analyze contribution management in LINUX, we first downloaded and mined the email archives of 131 LINUX kernel mailing lists between 2005 to 2009. We then used the MAILBOXMINER tool (Bettenburg et al. 2009a) to extract the mailing list data into a separate database that formed the basis for all our analyses. We carried out this intermediate step since raw mail archives in the form of mbox files are hard to process (Bettenburg et al. 2009a). The MAILBOXMINER tool is publicly available[8] under the GPLv3 license.

To enable the study of contribution management we developed a custom set of scripts to detect and extract contributions (which are submitted in LINUX in the form of patches) in the kernel mailing lists. Our tool is based on a set of regular expressions, similar to those employed in previous work of the Mining Software Repositories community (Bettenburg et al. 2008; Bird et al. 2007a). Based on this data, we extracted metrics on both the mailing lists (number of participants, frequency, volume, etc.), and the actual contributions themselves (size, files modified, complexity, etc.).

To obtain information on which contributions are accepted, we had to link the contributions in the emails to the distributed source control system of GIT following the approach of Bird et al. (2009). In particular, we developed a heuristic, which splits up a contribution per file that is being changed (this part of a contribution is called a "patch chunk"), extracts a list of files that are being modified by the contribution, removes noise such as whitespace, and then calculates a hash string in the form of a "checksum" of the relative file path and the changed code lines (Jiang et al. 2013). This is done to uniquely identify patches, as formatting changes widely across email (i.e., whitespace, line endings, word and line wrap). We follow the same approach to produce unique hashes for the accepted patches in the main LINUX GIT version control system. We then match contributions extracted from emails to contributions that were accepted into the main LINUX repository by comparing their hashes, and also taking into account the chronological aspects of email and commits (i.e., we do not link a patch that was accepted into the main repository if an email that contains a contribution with a matching hash was sent at a later date). Once this mapping from individual parts of contributions that found their way into the main LINUX source code repository to contributions that were submitted to the kernel mailing lists is established, we abstract back up from patch chunks (that describe parts of contributions) to user contributions.

We manually evaluated the performance of our linking approach on a sample of 3,000 email discussions that contain the identifiers of actual GIT commits, in which the discussions' patches were accepted. We measure the performance of our approach by means of "recall" (i.e., how many of those that we know were linked in reality are

[8] http://www.github.com/nicbet/MailboxMiner

also being linked by our approach). Upon manual inspection of a random stratified sampling of 100 emails (both linked and not linked) across all kernel mailing lists, we found that our approach had a precision of 100 % and a recall of 74.89 % on this sample. According to statistical sampling theory, this provides us with a 10 % confidence interval around our result at a 95 % confidence level (i.e. we are 95 % sure that the performance achieved with our approach lies between 90 % and 100 % precision). Both measures, precision and recall, provide us with confidence that the metrics we calculated on LINUX contributions are sufficiently accurate to obtain meaningful descriptions of the overall population of contributions sent over the LINUX kernel mailing lists.

Through these analysis steps we were able to recover a) what are the contributions that get sent over the LINUX mailing lists, b) who are the actors, c) which parts of the contributions get eventually accepted and d) all the surrounding meta-data such as dates, times, discussions on the contributions, as well as contribution size. These four parts of information form the main body of data for our study.

ANDROID The contribution management process of the ANDROID system is managed entirely through a dedicated server application, called GERRIT. GERRIT has a front-end user interface, which consists of a (client-side) web application running on Javascript.[9] Use of GERRIT is mandatory for any user who wants to submit a contribution to the ANDROID project. The user designs and carries out code changes in a local copy of the project repository. When done, the user needs to submit the entirety of the changes done as a delta to the main project repository. GERRIT, which sits on top of the distributed version control system GIT, monitors these submission requests, intercepts the request, puts it on hold and automatically opens a new code review task. Only when this code review has been approved and verified by a senior developer (as described in Section 2), does the contribution get sent on to the main project source code repository for integration.

The code review itself contains a large amount of meta-information surrounding the contribution management process, such as discussions on the proposed change, discussions on the source code, actors involved in the contribution and its review, as well as their votes in favour or against the contribution. In particular, the GERRIT system is designed to allow multiple people to vote and sign off changes in the distributed development system, as well as to inspect and comment on the source code itself.

According to an interview with Shawn Pearce, the lead developer of GERRIT, on FLOSS weekly,[10] the main goal of GERRIT is to provide a more formal and integrated contribution management system compared to an email-based approach found in projects like the LINUX kernel.

Since the GERRIT front-end like many other Google Web Toolkit (GWT) applications runs entirely as a client-side javascript program in the browser, classical approaches of mining the HTML pages for data (often referred to as "web-scraping") do not work. As a workaround, we created a custom mining tool that directly interfaces with the GERRIT server's REST services. The publicly-available source

[9] https://android-review.googlesource.com/

[10] http://google-opensource.blogspot.ca/2010/05/shawn-pearce-on-floss-weekly.html

code of the GERRIT system provides us with the necessary APIs to make REST calls to the server, retrieve data in the form of JSON objects and serialize these into Java Bean classes. We then copy the information contained in the Java Beans into a local database for further analysis. One advantage of this approach is that the Java Beans contain much richer data than what is already presented in the web-front end. In particular associations of entries with unique user ids, date precision and internal linking of artifacts are some of the highlights that greatly help our later analyses.

Overall, we obtained a snapshot of all the publicly visible contributions to the ANDROID project that were recorded through the GERRIT system from the start of the project, until July 2010. Our dataset contains 6,326 contributions from 739 active accounts that were contributed over the course of 16 months, starting from the initial open-source release of the ANDROID project. The GERRIT dataset contains:

(a) the source code of the individual contributions
(b) multiple versions of each contribution in the cases they needed to be revised
(c) discussions surrounding the contributions' source code (on a line-level)
(d) discussions surrounding the contribution management process including review and integration
(e) all votes that led to a decision on each
(f) meta-data such as dates, times, unique user ids, dependencies and other traceability links to related artifacts and repositories contribution granularity)

A manual inspection of a random sample of 100 entries suggests that the data GERRIT provides is both complete and clean, i.e., the "raw" data has all meta-data attached that we needed for this study, and no further cleaning or pre-processing steps were necessary.

2.2.2 Case Study on ANDROID and LINUX

In the following, we present commonalities and dissimilarities of the contribution management processes implemented in both projects. This presentation follows along the five phases of contribution management that we presented in our conceptual model (Section 2).

The analysis presented in this section forms the basis for our quantitative study of the contribution management processes of LINUX and ANDROID in the following section and highlights best practices in both projects.

Phase 1: Conception Both ANDROID and LINUX provide mailing lists for the conception and discussion of new ideas. The main difference with respect to contribution management is the traceability of a contribution from initial conception to final implementation. As ANDROID uses different technologies for conception (mailing lists) and all other phases (the GERRIT tool), the conception phase is separated from the remaining contribution management process, resulting in weak traceability.

LINUX on the other hand uses the same email discussion for both, conception, as well as review, thus enabling practitioners to follow a contribution from the initial idea to the final encoding in source code. The practices in LINUX have evolved historically: LINUX has been under development for almost 20 years and even though processes and technologies were adapted along the way (i.e., using different Version Control Systems to manage the project's source code repositories), the social process has mostly stayed the same.

ANDROID embraces modern web technologies (as opposed to LINUX), which are rooted in a more mature Web with advanced technologies that were not available 20 years ago and which have shaped how modern communities engage with online tools.

> *If traceability of contributions from initial conception of an idea to final implementation is a concern (e.g., for documentation or legal purposes), practitioners need to reconsider the usage of disjunct technologies in their contribution management process. One possible solution is the use of discussion forums that provide links in the form of unique discussion thread identifiers, which are referenced during any of the later phases of contribution management.*

Phase 2: Submission

Submission Channels: Contributions to ANDROID are submitted to the project's master version control system through a special purpose tool provided by the project. During the submission process a new peer review task is automatically opened in the GERRIT system that blocks the contribution from being delivered into the master version control system until a full peer review has been carried out.

In contrast, LINUX uses mailing lists for the submission of contributions. These mailing lists act as a repository that is detached from the version control system. Contributions are propagated to the version control system later on, once a contribution has been accepted in the peer review process of the submission.

Intellectual Property Management: In LINUX, community contributions are submitted through email and must contain a "signed-off" record that acts as a substitute for an electronic signature, to confirm with intellectual property regulations.

Intellectual property is made explicit in ANDROID, as contributors can only submit their contribution after they provide personal information by registering an account in the GERRIT system and signing a Contributor License Agreement (CLA) in the process.

> *If management believes that intellectual property management poses a risk, restricting submissions to registered users may be preferable.*

Phase 3: Review

Pending Review Notification: In ANDROID, each contribution triggers an automated notification to the project leaders and the appointed reviewers of the subsystem for which the contribution was submitted. This process is automated and handled by the GERRIT system. Reviewers can then access the new contribution through the web interface and make their thoughts known through either attaching a message to the general discussion of a contribution, or commenting directly on specific lines

in the contribution's source code. Either action triggers in return an automated notification to the original author of the contribution, as well as the other reviewers.

Peer review in LINUX is organized less formally. In addition to a project-wide mailing list for overall discussion, there exist many subsystem-specific mailing lists. Contributors are encouraged to submit their contributions through the corresponding mailing list of the subsystem that their contribution targets. However, functionally this is not much different from queues in GERRIT. Every community member subscribed to the subsystem mailing list sees new entries and can act on them as she sees fit. If a community member voluntarily decides to carry out a peer review, they can do so freely and at any time by sending their review as a reply to the corresponding email discussion.

Voluntarily managed peer review based on email is characterized by a significant amount of time that may pass before a volunteer provides first feedback (ref. Section 4-RQ1), and the risk that a contribution be completely ignored, when no volunteer steps up for review. An explicit notification-based strategy, such as the one employed by ANDROID, helps raise awareness of new contributions that require the attention of reviewers, and makes sure that every contribution is reviewed.

Judgement of contributions: In ANDROID, contributions are judged formally through positive and negative votes, cast by reviewers and verifiers. Only contributions that have at least one vote for acceptance (a +2 vote) can move on to verification. In particular, reviewers in ANDROID can cast a +1 vote to indicate that the contribution should be accepted, and a −1 vote to indicate that the contribution should be rejected. Senior members, i.e., verifiers, then assess the votes of the reviewers and decide on acceptance of the contribution (they cast a +2 vote), or rejection (they cast a −2 vote).

The judgement of contributions in LINUX is less formal. Contributions are either *abandoned*, if the community showed not enough interest (with respect to follow up emails on the original submission); *rejected*, if a project leader decides that there were too many concerns raised by the community; *revised*, if there were only minor concerns that could be corrected easily; or *accepted* as is. In contrast to ANDROID, acceptance of a contribution in LINUX is implicit: a contributor knows whether his contribution was accepted, when the maintainer of a subsystem accepts the contribution in his own copy of the master version control system. In case of revision, updated versions of the contribution are commonly submitted to the same email thread.

If assessment of contributions by multiple reviewers is a concern, a voting-based approach, as employed by ANDROID, provides transparency and documents the decision making process. Such a voting-based approach might be preferable over a hierarchical approach, like the one employed by LINUX, where decisions can be overthrown at any time by individuals at a higher level in the project hierarchy.

Phase 4: Verification In ANDROID, verifiers, who are often senior (Google) engineers, who have been appointed by the leaders responsible for the individual ANDROID subsystems, merge the contribution into a local snapshot of the latest version of the code base, and manually test the contribution for correctness and functionality. If problems occur, the verifiers give feedback to the original contributor, who can then resubmit an updated revision that addresses these problems. If verification succeeds, the contribution is accepted to ANDROID's development branch by the subsystem's lead reviewer or maintainer.

In LINUX, verification of the contribution takes place through developers and beta testers of the experimental branch. Feedback is provided to the original contributor through separate mailing lists that are dedicated to this testing purpose. If problems occur during this phase, contributions are put on hold until the issues are resolved.

> *If workload on in-house developers is a concern, outsourcing of review and verification effort to the community can help reduce overall workload on senior developers. However, this comes at the cost of giving up control over the final quality assurance, and can lead to even longer processing of contributions.*

Phase 5: Integration In ANDROID, integration of contributions takes place as soon as they have been successfully verified. Project leads can initiate an automated integration process through the web interface of the code review system. If this automated merge is successful, the contribution is delivered to the community the next time they synchronize their local environments with the project directory.

Integration of contributions into the next release in LINUX is handled through a semi-automatic approach: contributions are manually integrated by developers, by moving them through development repositories until they finally reach the main development branch (Jiang et al. 2013). However, to have a chance for being integrated in the upcoming version, a contribution has to be at least accepted during the merge window of the upcoming release. But even then, contributions can fail to make it, for example if the contribution is too risky to introduce at once or other features suddenly get higher priority. Linus Torvalds has the final say ("benevolent dictator") (Crowston and Howison 2005) and can overrule all previous recommendations to reject contributions that do not fit with the strategic direction of LINUX.

We observe that in both systems some dedicated developers have the role of decision makers, who can ultimately deny the integration of a contribution into the master repository. ANDROID's integration strategy is much more immediate than the strategy selected by LINUX, and delivers accepted contributions to the rest of the community without delay.

> *We observe that both LINUX and ANDROID make use of semi-automated tools, which sit on top of the master source code repository, and offer immediate integration of contributions and delivery, as well as feedback to the contributor, in case the integration process fails.*

3 Quantitative Assessment of Contribution Management in ANDROID and LINUX (Q2–Q4)

In the second part of this paper, we present a quantitative assessment of the contribution management processes of ANDROID and the LINUX kernel. For this, we follow along our remaining three research questions:

Q2: To what extent are community members actively engaged in the development of the project?
Q3: What is the size of received contributions?
Q4: Are contributions managed in a timely fashion?

For each research question, we first present a motivating discussion, and then contrast our findings on the contribution management of both projects along individual characteristics. We highlight our main findings and lessons learned during quantitative and qualitative analysis steps in boxes towards the end of each part. Throughout our quantitative study, we describe our findings through descriptive statistics. In particular, we are using the arithmetic average (mean) as a default measure of centrality. However, some of the data we report on is not normally distributed, i.e., data such as the lifespan of community members is heavily skewed with long tails. In these cases, the mean would in turn be a skewed and thus inappropriate measure of centrality. In these cases we report the median, rather than the mean to present a more accurate descriptive statistic suitable for skewed distributions.

3.1 Are Community Members Actively Engaged in the Development of the Project? (Q2)

The profound value of a community surrounding a software product for open source business models is captured in two principles: Reed's "law of the pack" and Bass' "diffusion model" (Bass 1969; Reed 2001). Based on Reed's law, we conjecture that the size of the community surrounding a software product (often referred to as "product halo"), is an indicator of the project's success and business value. A similar concept is captured in Bass' diffusion model, which states that *"As more people get involved in a community, participation begets more participation"*. However, the different contribution management strategies employed by LINUX and ANDROID might have an effect on the extent to which the community is engaged in the development process.

To answer this research question, we measured the overall amount of contributors in ANDROID and LINUX, as well as the overall time during which contributors actively participated in the development of the software product. We use these measures as a proxy to judge whether both projects are successful in engaging and motivating external developers to take part in a collaborative process surrounding the software products.

In the following, we refer to the state of LINUX during the year 2005 (the earliest data after the most recent switch of their contribution management process) as LINUX 05 and to the state of LINUX during the year 2009 (the same time period for which we collected ANDROID data) as LINUX 09. We split the LINUX dataset into two parts to make comparisons between LINUX and ANDROID fairer. In particular, LINUX 05 describes the first year of LINUX' current contribution management

Table 2 Results of quantitative analysis of community activity in ANDROID and LINUX

Metric	ANDROID	LINUX	
		2005	2009
Number of community members	739	2,300	4,901
Number of reviewers	408	1,680	3,503
Number of contributors	526	771	2,482
Number of contributors with multiple submissions	203	475	1,792
Number of contributors with multiple submissions who had at least one rejection	151	445	1,666
Number of contributors who had a single submission that was rejected	37	208	405
Number of contributors who returned to submit more contributions	38.5 %	61.6 %	72.19 %

process, and similarly, the ANDROID dataset describes ANDROID's first year of contribution management. For both datasets, we collected data up to the following of the reported year, and for counting returning contributors in particular have followed through to that point (e.g., for ANDROID, we report on data until January, but looked until July to see if those commits' developers sent anything later on).

Part 1: Size of the Community We present the size of the community surrounding the development of ANDROID and LINUX in Table 2 and Fig. 2, as counted from the unique authors and reviewers of contributions in the datasets that we collected from the LINUX kernel mailing lists and the GERRIT code review system. We measured a total of 739 active community members in ANDROID, of which 408 have carried out peer reviews and 526 submitted contributions. For LINUX 05, we measured a total of 2,300 active community members, of which 1,680 have carried out peer reviews and 771 submitted contributions. For LINUX 09, we measured a total of 4,893 active community members, of which 3,503 have carried out peer reviews and 2,482 submitted contributions.

From a management point of view, returning contributors are more valuable than first-time contributors, since they are already familiar with the project, coding

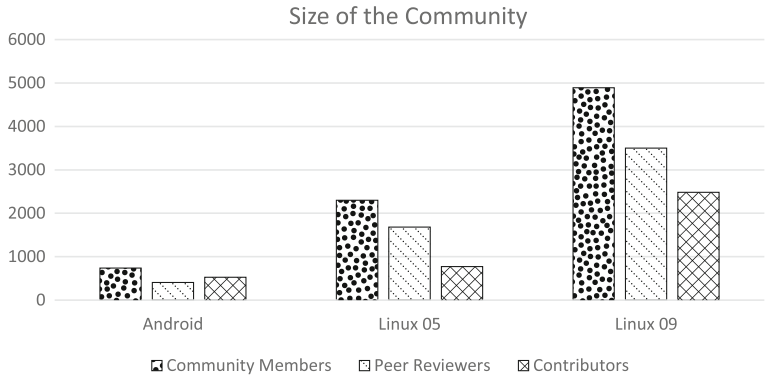


Fig. 2 Comparison of community size in ANDROID, LINUX 05 and LINUX 09

policies and processes. In ANDROID, 38.5 % of contributors returned to submit more contributions. In LINUX 05 and LINUX 09, these returning contributors rate is much higher: 61.6 % in 2005 and 72.19 % in 2009. Overall, LINUX was successful in increasing the returning contributor rate by 10.6 % over the course of four years.

Overall, we observe that both projects are able to attract sizeable and growing communities with 38 % (Android) to 62 % (Linux) of community members contributing multiple times.

Part 2: Lifespan of Community Members We measured the activity of community members as the time span between the date of their first activity (contribution, message, review, or comment) until the date of their last recorded activity. We leave out those community members for which only a single activity was recorded and ignore any possible hiatus between periods of activity (i.e., a contributor's period of activity begins with the date of submission of that contributor's first submission, and ends with the date of the last recorded submission).

We find that the median period of activity of community members in ANDROID is 65 days. In LINUX 05, community members have a median period of activity of 24 days, whereas in LINUX 09, the median activity is 57 days.

To get an idea of the potential impact of the lifetime of community members on contribution management, we carried out a manual inspection of all ANDROID contributions that were submitted by one-time contributors. We found that 19.51 % of these contributions were abandoned because contributors were no longer available to process the feedback given by reviewers, as reviewers took longer than two months (= 60 days) to give feedback on the contribution. The remaining 80 % of abandoned contributions were due to a variety of reasons, such as:

- the contributor implemented functionality that was already implemented in-house, but in the non-public development branch. As the development branch is not publicly available, the contributor was not aware of this duplication. (48.7 %)
- the contributor implemented functionality that was already implemented by another contributor at the same time. (7.3 %)
- the contribution conflicted with the master source code repository beyond feasible repair. (4.8 %)
- the contribution was submitted to the wrong subsystem. (7.3 %)
- the contribution was deemed incomplete or of too low quality. (36.6 %)

As a result, we conjecture that an effective contribution management process needs to be aware of the time span in which community members are active and available for feedback and carrying out possible changes to their initial contribution. When first feedback is given to contributors after they have already left the project, contributions often end up in an unfinished state and are ultimately abandoned, thus wasting precious time of both contributors and reviewers that could have been directed elsewhere.

3.2 How Large are Received Contributions? (Q3)

One of the major benefits when moving towards an open source business model is free, outsourced development, contributed by the community. However, past

research has reported that on the one hand contributors often submit small, insubstantial contributions (Weissgerber et al. 2008), and on the other hand peer review processes also tend to favor small changes (Rigby et al. 2008) as they are easier to work with. However, we believe that many companies prefer large contributions that are split into smaller chunks, as individual chunks are easier to review and processing of the entire contribution can be distributed, and thus be carried out more efficiently.

Approach To answer this research question, we first measure the size of ANDROID and LINUX contributions, as well as the number of files modified by contributions. We then split up contribution size for accepted and rejected contributions to examine whether acceptance is biased towards larger or smaller contributions (or in other words: whether larger or smaller contributions are favoured).

Since it is hard to distinguish modifications of source code lines from added and removed lines, we define the size of a contribution as the sum of the added and removed lines in the patch output of ANDROID and LINUX contributions (Rigby et al. 2008). As a result, our measurements basically consider the sum of actual added lines, actual removed lines and twice the number of modified lines.

Part 1: Contribution Size and Spread The median size of contributions in ANDROID is 16 lines, with 75 % of the contributions smaller than 57 lines. In LINUX 05, the median size is 36 lines (75 % of the contributions smaller than 187 lines), compared to a median size of 38 lines for LINUX 09 (75 % of the contributions smaller than 207 lines). For both projects we observe that the average contribution is relatively small, but occasionally, very large contributions are received (i.e., observed in the long tails of distributions in Fig. 3).

To give a point of reference, we counted the size of files in lines of code (LOC) for two specific versions of both projects that fall into the timeframe of our analysis. We find that the average number of lines of code in files of LINUX kernel version 2.6.17 is 21.46 LOC; and the average numbers of lines of code in files of the ANDROID version 2.2 ecosystem is 41.0 LOC.

We observe that while 50 % of the ANDROID contributions change a single file (75 % are spread across less than 4 files), we find that 50 % of contributions in both, LINUX 05 and LINUX 09, are spread across 2 files (75 % across less than 7 files). We present box-plots of the spread (the number of files changed by a contribution) in Fig. 4.

Overall, we note that both projects received substantial contributions from the community. Even though the integrated contribution management system used by the ANDROID project (GERRIT) offers visualizations of submissions on file and line level to developers, the manual reviewing approach in LINUX appears to be able to effectively cope with submissions that are more than twice as large, and spread across double the number of files.

Part 2: Acceptance Bias To analyze the relation of contribution size to submission acceptance, we classified all contributions to both projects into two groups (ACCEPTED, and REJECTED) and carried out a statistical analysis of the resulting distributions (Fig. 3). The median size of accepted contributions is 16 lines of code for ANDROID, compared to 11 lines of code for rejected contributions. Similar

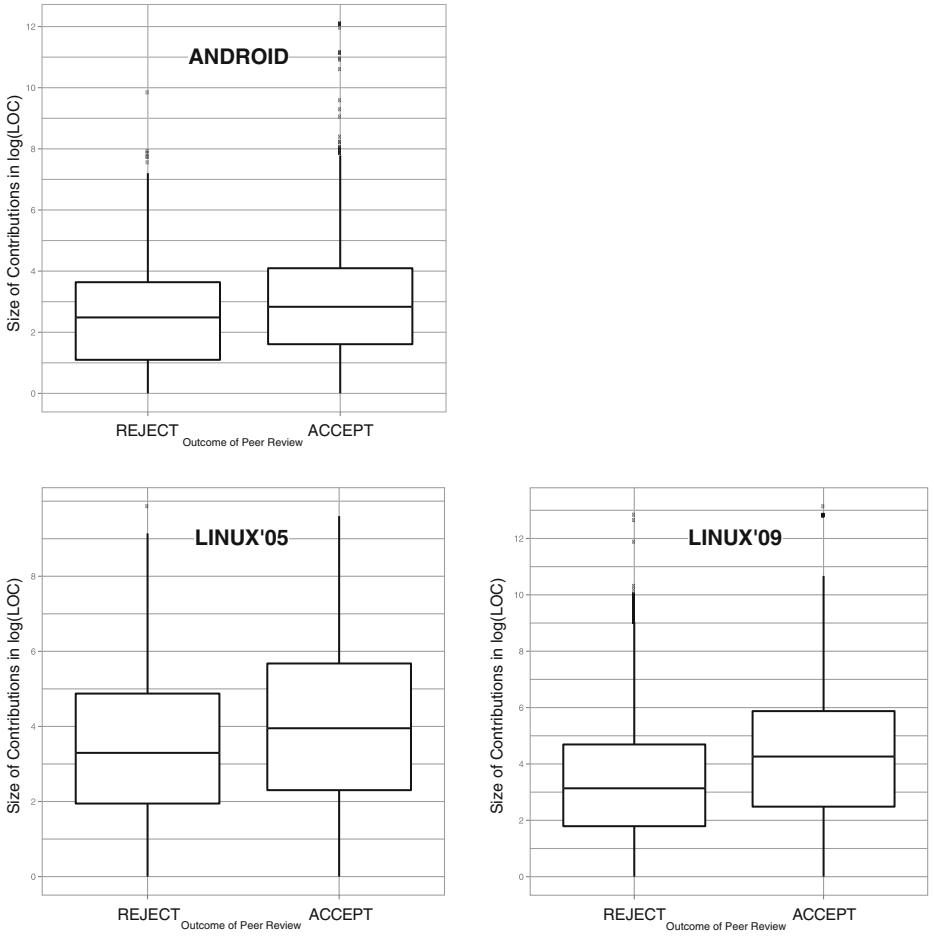


Fig. 3 Size of contributions (in log(Lines of Code)) with respect to final review outcome

differences exist for LINUX 05 (52 lines of code for accepted vs. 27 lines of code for rejected submissions) and LINUX 09 (71 lines of code for accepted vs. 23 lines for rejected submissions).

We performed a non-parametric t-test (Mann–Whitney–Wilcoxon) on each dataset independently, and reject our null-hypothesis H_0, *"There is no difference in size between accepted and rejected contributions"*, at $p < 0.001$, or in other words, the difference in contribution size between accepted and rejected contributions is statistically significant.

Overall, we find that for both projects, accepted submissions are between 1.6 and 3.0 times larger than rejected submissions. This stands in contrast to previous research in the area, which has demonstrated that smaller contributions are favored in peer reviews (Rigby et al. 2008). Our data shows that decisions on the final outcome of acceptance or rejection of community contributions in LINUX and ANDROID follow opposite patterns.

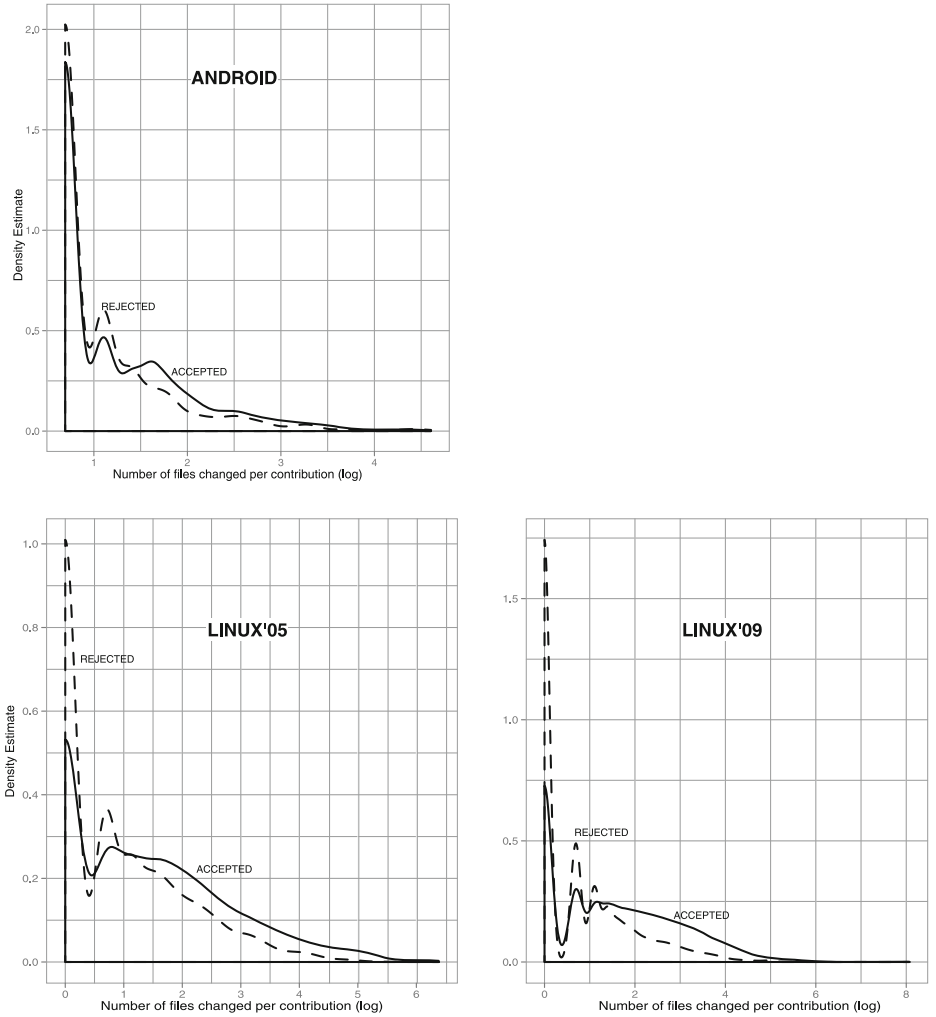


Fig. 4 Spread of contributions (in log(number of files changed)) with respect to final review outcome

We observe that both projects received substantial contributions from the community. In both projects we observed a statistically significant acceptance bias towards larger contributions.

3.3 Are Contributions Managed in a Timely Fashion? (Q4)

A good contribution management process needs to be able to manage contributions in a timely fashion to avoid potential negative impacts on the development process. As examples to motivate this research question consider the following scenarios.

(1) A contributor might become neglected through an overly long wait for his contribution to be addressed and leave the community.

(2) An internal development team might spend effort on the same task that has already been contributed by the community (and is still awaiting review).

Approach To answer this research question, we analyzed the delays introduced to the contribution management process through the review phase. We did not consider design discussions, as conception and maturing of ideas into source code submissions happen independently and cannot actively be influenced (apart maybe from moderating discussions). We neither considered verification nor integration, as both projects follow very different approaches that we cannot directly compare (as opposed to the review phase). In literature, peer reviews have been demonstrated to require significant scheduling and inspection effort from developers (Johnson 1998; Porter et al. 1998; Rigby et al. 2008), and are hence a major factor for the overall turnaround time of a contribution.

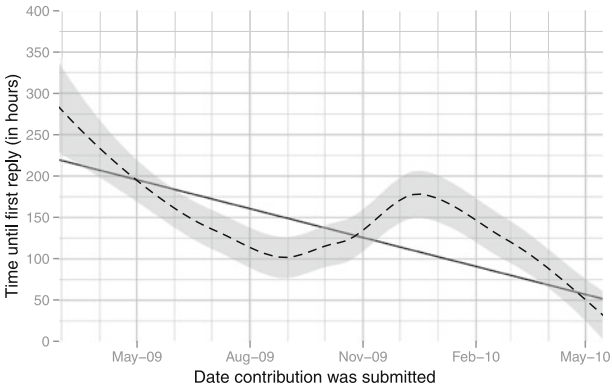
While conceptually there is a "return for rework" phase, i.e., when a contribution is flawed, the contributor might be asked to update the contribution according to the review board's comments and re-submit, these are not marked as such explicitly in the data (neither for ANDROID, nor for LINUX). We did not find a reliable way to automatically distinguish a return-for-rework state, and furthermore we observed a broad set of reasons for submissions of updated patches within the same thread. However, we capture the idea of "first feedback" to the author, which is a superset of return-for-rework.

We looked at the temporal properties of the peer review phase from two different angles: after submission, contributors are mostly interested in the time it takes to get initial feedback on their contribution, whereas the reviewers and project leads are mostly interested in the total time it takes to carry out the peer review phase.

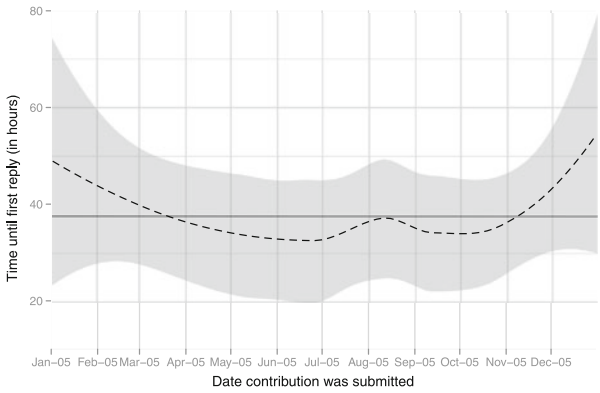
For each project, we first recorded for each contribution the submission date, the time in hours until a first response was recorded, and the overall time in hours until reviewers announced their final decision. If there was never a response recorded in the data, we ignored that contribution for this experiment. We then fit a linear regression model (Dobson 2002) (e.g., shown as a solid line in Fig. 5) to observe overall trends, and a LOESS polynomial regression model (Cleveland and Devlin 1988) (e.g., shown as a dotted line in Fig. 5). The choice of fitting a LOESS curve to the data enables us to observe local trends in the data, such as seasonal shifts that would otherwise be missed by a linear model, which tells us the general direction—or trend of the data.

Part 1: Time Until First Response For ANDROID, we find that the average time until initial feedback is given *decreased* by a factor of 3.55, from 221.91 h (approximately nine days) for contributions submitted in March, 2009 to 62.37 h (approximately two and a half days) for contributions submitted in March, 2010. In contrast, we find that for LINUX, the average time until initial feedback *increased* by a factor of 1.37, from 37.63 h (one and a half days) in 2005 to 51.40 h (about two days) in 2009.

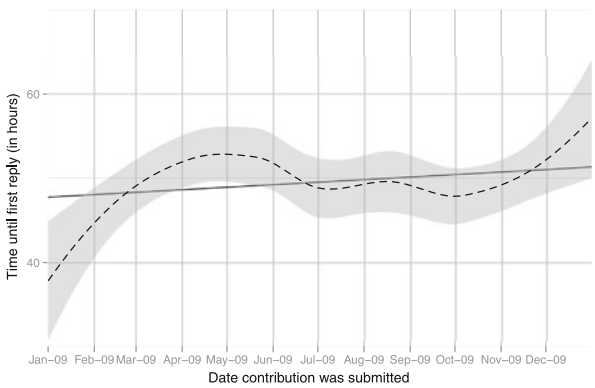
Part 2: Overall Time Taken for Peer Review In LINUX, the average time needed to complete the review phase *increased* by 7.2 % from 183.80 h (approx. seven and a half days) in 2005, to 197.90 h (approx. eight days) in 2009. In contrast, we find that for ANDROID, the average time to complete the review phase *decreased significantly* from 522.20 h (about 21 days) in March 2009 to 80.34 h (about 3 days) in March 2010. As we discuss in the following, this decrease is largely due to two effects: first, the



(a) ANDROID



(b) LINUX 05

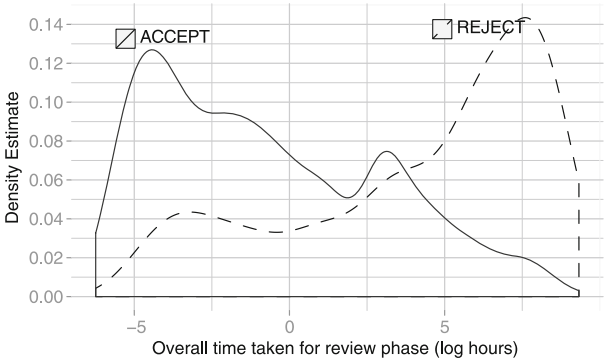


(c) LINUX 09

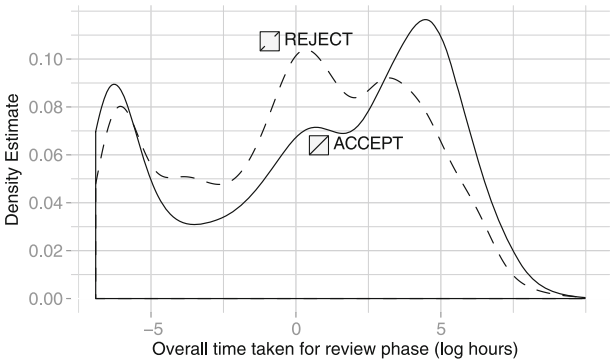
Fig. 5 Time until a first response for a submission is received during the review phase

practice of performing clean-up before a release of ANDROID, and second, active efforts in decreasing feedback delays.

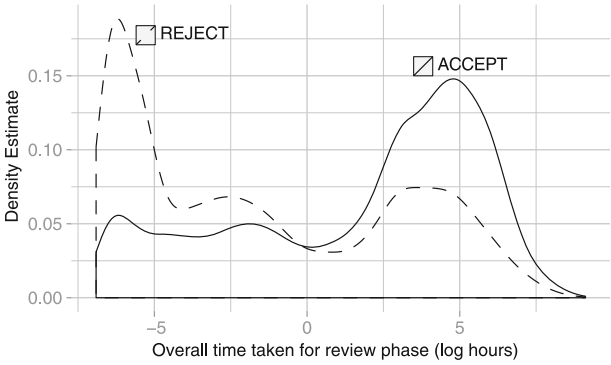
In addition, we investigate the relation between overall review time and outcome through kernel-density analysis (Rosenblatt 1956). The plots derived from this analysis are presented in Fig. 6.



(a) ANDROID



(b) LINUX 05



(c) LINUX 09

Fig. 6 The overall time taken to reach a decision differs significantly across projects

Our kernel density plots are estimates of the probability density functions of the random variables connected with acceptance or rejection of contributions. The actual values of the y-axis in these cases depict the probability of the random variable attaining the value at the corresponding point on the x-axis. Within this context, kernel density plots should be viewed as very precise histograms. The big advantage is that histograms can appear greatly different depending on the number of bins

an analyst specifies and easily over or under-sample the data at hand, while kernel density estimates automatically derive an optimal bandwidth.

Our observations show that reviewers in ANDROID are fast in deciding whether to accept a contribution, but take much more time to reject a submission (Fig. 6a). Most considerably, we observe the opposite for LINUX. From 2005 to 2009, decisions on whether to reject a contribution take up increasingly less time, and decisions about accepting a contribution take up increasingly more time (Fig. 6b and c).

In LINUX, we observe that contributions are rejected quickly, yet a decision for acceptance takes considerably more time. In ANDROID, we observe that contributions are accepted quickly but a final decision towards rejection takes much longer.

Discussion of Parts 1 and 2 A possible explanation for the observed increase in overall review time, as well as feedback time from LINUX 05 to LINUX 09, might be a decreased ratio of reviewers to contributions, as well as an increased submission volume per contributor.

To study this hypothesis, we categorized community members into two classes, contributors and reviewers. We consider a community member as a contributor, if our data contains at least one submission from this member. We consider a community member as a reviewer, if our data contains at least one peer review activity from this member. Since members can assume both roles at the same time, we account for this overlap by assuming that contributors will not review their own contributions (this is a strong assumption that might not hold true in reality, but it makes counting more feasible).

For LINUX 05, we measured the number of reviewers of community contributions and the number of contributors from our dataset and find that for each contributor, there are 2.17 reviewers, and that every contributor submits an average of 12 contributions (median: 2). Similarly, for LINUX 09 we find that for each contributor there are 1.41 reviewers, and that each contributor submits an average of 28.68 submissions (median: 4).

While the increase of feedback delay in LINUX can be explained by an increase in per-contributor submissions on the one hand, as well as a reduction of available reviewers per contribution, we found no such evidence for ANDROID.

However, a plot of the raw data of response times for ANDROID presented in Fig. 7 reveals a series of triangular patterns. These patterns indicate the early struggles of the ANDROID project to give timely feedback: before June 2010, the established practice was to batch-process all contributions in the system around the time of a major release (October 2009 for ANDROID Eclair 2.0 and May 2010 for ANDROID Froyo (2.2)).

The recorded feedback times lie on an almost perfect slope that meets the x-Axis at the major release dates. This practice led to a variety of problems, such as abandoned contributions due to a lack of interactivity between senior members who judged contributions and the contributors (who had already moved on and were no longer actively engaged in the project). In June 2010, Jean-Baptiste Queru, one of the lead developers of ANDROID announced a radical change of the review process, as documented on the ANDROID development blog[11]

[11] http://android-developers.blogspot.com/2010/06/froyo-code-drop.html

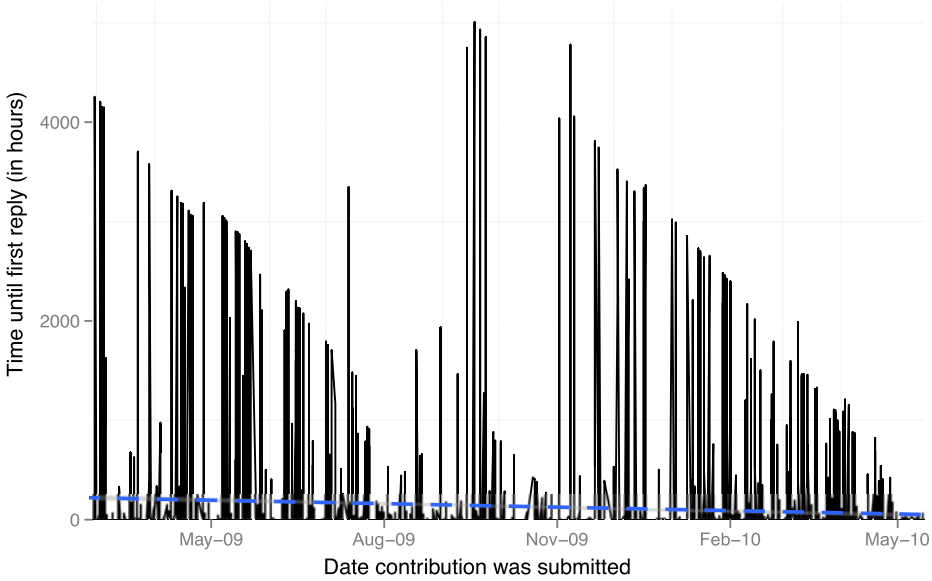


Fig. 7 The time until a contributor is given first feedback on a submission in ANDROID

> *"We're now responding to [ANDROID] platform contributions faster, with most changes currently getting looked at within a few business days of being uploaded, and few changes staying inactive for more than a few weeks at a time. We're trying to review early and review often. [...] I hope that the speedy process will lead to more interactivity during the code reviews."*

Upon further analysis of the ANDROID dataset, we observed that distributions of the review times are biased towards rejection (ref. Fig. 8). A common practice in ANDROID is to perform clean-up of the GERRIT system right before a software release. As part of that clean-up phase, contributions that have been open for a long time and that are not actively pursued, e.g., a reviewer has been waiting for the contributor to submit an updated version of the patch for an extended period of time, are closed with a rejected status.

Even though we observe a three-fold increase in response time in LINUX, the results of our analysis show that the impact on overall review time is relatively small (ca. 7 %). This finding suggests that, while the self-managed appointment of reviewers in LINUX is suited for absorbing a large increase in submission volume without causing an overall increase in review times, initial feedback suffers. One possible explanation could be that a large increase in contribution volume also increases the amount of email messages community members receive, and thus more choice when picking contributions for review.

The ANDROID project switched from a periodical batch-processing of contributions around the time of major releases to a process that allows for early feedback and frequent reviews. This change in process was purposely done to increase interactivity between senior developers and contributors during the review phase of the contribution management process. Through the case study on LINUX, we documented the need of contribution management processes to account for increases in submission

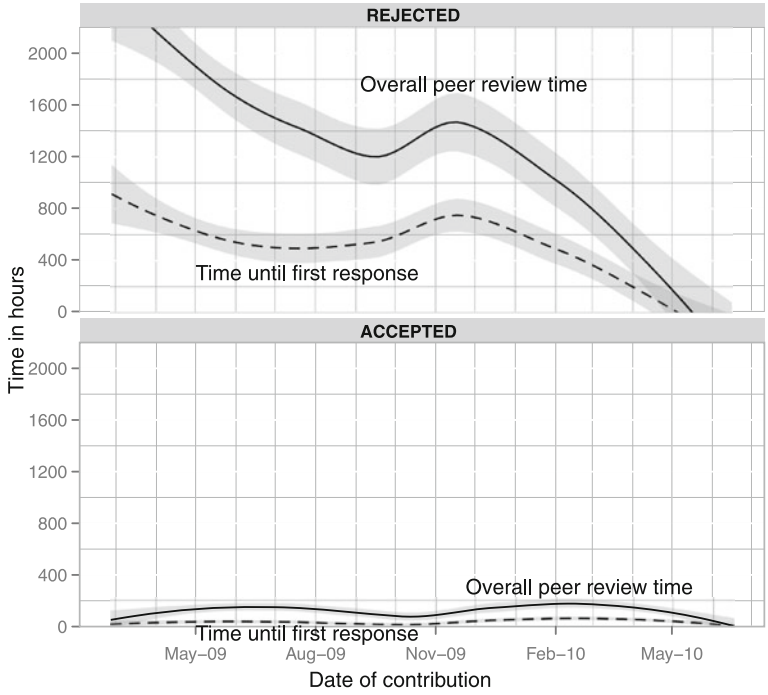


Fig. 8 Feedback and review time in ANDROID by final decision outcome

volume and community size to avoid delays. This is especially important to avoid losing valuable contributions due to contributors having only a short timespan in which they are active and available for feedback.

4 Threats to Validity

In the following we discuss the limitations of our study and the applicability of the results derived through our approach. For this purpose we discuss our work along four types of possible threats to validity (Yin 2009). In particular, these are: construct validity, internal validity, external validity and reliability.

4.1 Construct Validity

Threats to construct validity relate to evaluating the meaningfulness of the measurements used in our study and whether these measurements quantify what we want them to.

Our study contains a detailed case study of contribution management in ANDROID and LINUX. Within that case study, we quantify key characteristics of contribution management, along three dimensions, in particular, each dimension corresponds to a particular research question.

Within each dimension (activity of the community, size of contributions, timely management of contributions), we selected multiple measures that highlight the

studied dimension from different angles. Our quantitative assessments are based on descriptive statistics about the distributions of the collected data samples. When comparing findings across both projects, we support these comparisons through statistical hypothesis testing.

Time-span and trend data is studied through fitting of two regression models, first a linear regression model to observe the overall trends of the data in the studied time periods, and second a polynomial regression model to account for possible seasonal variation in our data, i.e., to counter bias introduced by software release cycles and software development processes in both projects.

In all cases where we performed multiple statistical significance tests, we used the Bonferroni correction (Rice 1995) to avoid the spurious discovery of significant results due to multiple repeated tests.

With respect to the conceptual model of contribution management presented in the context of research question 1, threats to construct validity concern the extent to that our observations match reality. The conceptual model was systematically derived from multiple significant open-source software projects. All seven projects that were analyzed showed significant commonalities of managing contributions in a series of steps. Our conceptual model generalizes these steps into five distinct phases that contributions undergo in each project, before they become part of the software and are made available to the general public.

Within each of these phases however, there may exist processes and practices that are specific to a particular project. Our work presents two instances of the conceptual model which documents these processes and practices for two major open-source systems, LINUX and ANDROID. While we cannot claim completeness of the model (i.e., perfectly fitting the contribution management process of every open-source project in existence), we see our model as a first starting point for documenting and formalizing contribution management, in the same vein as architectural models have been derived and refined in the area of Software Architecture research.

4.2 Internal Validity

Threats to internal validity relate to the concern that there may be other plausible hypotheses explaining our findings.

We have carried out a detailed case study on how two related open source systems, ANDROID and LINUX do contribution management in practice. While ANDROID and the LINUX kernel share many similarities, for example drivers, device support, and core operating-system functionality that is added to the systems—we could argue that to some extent ANDROID contains LINUX. Even though both projects share similar roots, mindsets, tools, processes and domain, we believe that a comparison, such as carried out in our case study, is worthwhile and insightful. Our main goal is to study the management of contributions, rather than technical aspects or implementation details of the OS software.

We want to note that the scope of our study is not the technical aspects (or even content) of the individual contributions, but how contributions are managed in practice. With that respect, LINUX has switched to their current contribution management practices at about the same time as the development in ANDROID started. From documentation (e.g., the interview with Shawn Pearce referenced in our paper) we observed that ANDROID had looked closely at how LINUX manages contributions,

and they consciously decided to pursue a different approach of contribution management that better suits their business.

Within our detailed case study on contribution management, we quantify and compare key characteristics of LINUX and ANDROID. Our quantitative findings and resulting hypotheses regarding the cause of these findings were followed up by detailed manual and qualitative study of the underlying data, to balance the threats for internal validity of our study.

4.3 External Validity

The assessment of threats to external validity evaluates to which extent generalization from the results of our study are possible.

In this study, we propose a conceptual model of contribution management. This model is an abstraction of the (commonalities in) contribution management processes of seven major open source projects. In the same way that architectural models create abstractions of actual instances of software implementations, our goal is to give researchers a common ground for conversation about, and further study of contribution management.

While the conceptual contribution management model was derived from only a tiny fraction of open source projects in existence, we still argue that generalizability of the model is high. The conceptual model was derived through a systematic approach, known as "Grounded Theory" (Glaser and Strauss 1967; Strauss and Corbin 1990), of publicly accessible records of processes and practices of seven projects contemporary, prominent and important open-source projects.

While we can claim neither completeness, nor absolute correctness of the derived model, our abstraction serves as a starting point for an abstraction of contribution management on a scientific basis, and we hope that future research will lead to incremental refinements of this abstraction, similar to conceptual models of software architecture.

Our detailed case study on LINUX and ANDROID, illustrates real instantiations of contribution management in practice, and details key characteristics along three dimensions. Due to the nature of this study, our observations are bound to the two studied systems, and unlikely to generalize to the broad spectrum of open source projects in existence. However, we report on a variety of practices that are found in many open source projects, such as "cherry-picking", i.e., selecting only small parts of a contribution for inclusion into the project, and outline problems, for example, the re-implementation of functionality by multiple contributors at the same time. As such our case study stands as a report of examples in "best-practices" and potential "pitfalls" in two large and mature open source systems, which are likely to be applicable across a broader range of domains and other open source projects.

4.4 Reliability

The assessment of threats to the reliability of our study evaluates the degree to which someone analyzing the data presented in this work would reach the same results or conclusions.

We believe that the reliability of our study is very high. Our conceptual model of contribution management is derived from publicly available documentation and data

of seven large open-source software systems. Furthermore, our case studies on how the LINUX kernel and ANDROID OS projects carry out contribution management in practice rely on data mined from publicly available data repositories (email archives in the case of LINUX, and GERRIT data in the case of ANDROID). The methods used to collect that data are described in detail in Section 2, and have also been used in previous research, e.g., the work by Jiang et al. (2013).

5 Related Work

The work presented in this paper is related to a variety of previous studies on open source development processes, which we discuss in the following.

5.1 Community-Driven Evolution Through Code Contributions

In their work "The Cathedral and the Bazaar", Raymond (2001) discussed core ideas behind the success of the open source movement. Raymond's main observations are formulated as Linus' law, i.e., the more people reviewing a piece of code, the more bugs can be found, and on the motivation of developers to add the features they are interested in.

While our work does not attempt to prove or disprove Linus' law, we have studied two systems that strongly support and encourage user-driven evolution, but are substantially different in the way they treat user contributions and merge them into their own code-bases. For instance, in ANDROID, contributions do need to align with the strategic goals of the leaders for the module these contributions target, a community interest in a feature alone is not sufficient for inclusion.

For example, ANDROID community contribution #11758 was rejected despite large community interest (in ANDROID, users can "star" a proposed contribution if they are enthusiastic about having that feature added) for the contributed feature, in particular one reviewer of the contribution[12] notes:

> *"It might have 40 stars but you have to weigh in the cost of adding a rather obscure/technical UI preference for *everybody* for the benefit of a few."*

5.2 Community-Driven Development as a Business Model

Both Hecker (1999), and Krishnamurthy (2005) provided a comprehensive overview and analysis of modern open source business models. While Hecker outlined potential pitfalls that businesses have to be aware of when moving towards these models, Krishnamurty described key factors for the success of open source business models.

In particular, Hecker et al.'s work (Hecker 1999) puts a large emphasis on the importance of the community in the open source business model. Our work extends on previous knowledge through our qualitative study on how two major and successful open source projects handle business concerns like intellectual property management, peer review, and the potential risk of meaningless contributions.

[12] https://android-review.googlesource.com/#/c/11758/

5.3 Community-Contribution Management

Mockus et al. (2002) investigated email archives and source code repositories of the APACHE and Mozilla projects, to quantify the development processes and compare them to commercial systems. They found that APACHE has a democratic contribution management process, consisting of a core group of developers with voting power and CVS access, and a contributor community of 400 developers. In addition, Mockus et al. identified Mozilla as having a "hybrid" process, since it was spawned from the commercial Netscape project.

Furthermore, Mockus et al. conjectured that "it would be worth experimenting, in a commercial environment, with OSS-style open work assignments". Our work extends on this notion through the systematically derived coneptual model of contribution management, as well as our qualitative and quantitative investigation of two concrete instances of that conceptual model.

Rigby et al. (2008) investigated peer review practices in the APACHE project, and compared these to practices observed in a commercial system. They found that small, independent, complete contributions are most successful, and that the group of actual reviewers in a system is much smaller than potentially is achievable. The work by Rigby et al. focused on the time component of reviewing large and small contributions, while our work investigates acceptance bias.

Capiluppi et al. (2003) studied the demography of open source systems, and found that few projects are capable of attracting a sizeable community of developers. In particular, Capiluppi et al. found that 57 % of the projects consist of 1 or two developers, with only 15 % having 10 or more developers, leading to slow development progress. In addition, Capiluppi et al. remarked that larger projects typically have 1 co-ordinator for every 4 developers. In contrast to their work, we find that for both LINUX and ANDROID there is a significantly higher ration of co-ordinators to developers.

Weissgerber et al. (2008) studied patch contributions in two open source systems, and found that 40 % of the contributions are accepted. Contrary to our findings, they find that smaller patches have a higher probability of being accepted.

Crowston and Howison (2005) examined 120 open source project teams, and found that their organization ranges from dictatorship-like projects to highly decentralized structures. Our work presents two instances of organizatorial structure: LINUX as a hierarchy of individuals with increasing power over the ultimate decisions connected with community contributions, and ANDROID, as a decentralized, vote-based structure.

Within the same vein of the work presented in this study, Sethanandha et al. (2010) proposed a framework for the management of open source patch contributions which they derived from 10 major open source projects. While the work of Sethanandha et al. focussed more on the handling of patches and their application against the code base, our work paints a more general picture of contribution management, which spans from inception to integration of a contribution. In addition, we carry out a detailed case study on two open source ecosystems to investigate how the process is implemented in practice.

6 Conclusions

Contribution management is a real-world problem that has received very little attention from the research community so far. Even though many potential issues with accepting contributions from external developers (i.e., developers who are not part of an in-house development team) into a software project have been outlined in literature, little is known on how these issues are tackled in practice. While deriving a conceptual model of contribution management from seven major open source projects, we found that even though projects seem to follow a common set of steps—from the original inception of a contribution to the final integration into the project codebase—the different contribution management practices are manifold and diverse, often tailored towards the specific needs of a project.

Even though both studied systems (LINUX and ANDROID) employ different strategies and techniques for managing contributions, our results show that both approaches are valuable examples for practitioners. However, each approach has specific advantages and disadvantages that need to be carefully evaluated by practitioners when adopting either contribution management process in practice. While a more open contribution management process, such as the one employed by LINUX, reduces the overall management effort by giving control over the process to a self-organized community. Disadvantages of such self-organized and community driven contribution management include weaker intellectual property management, and the risk of missed contributions. A more controlled contribution management system, such as the one employed by ANDROID overcomes these advantages, at the cost of an increased management effort that, depending on the size of the community, might burn out the reviewers, since they are forced to review changes and follow up on eventual revisions.

The findings of our quantitative assessment of the contribution management processes of both projects, makes a case for the importance of timely feedback and decisions to avoid losing valuable contributions. In contrast to LINUX, where even a more than three times increase of feedback delay does not seem to be a cause for alarm, we found that ANDROID makes active efforts to decrease feedback times, with the goal to foster increased interactivity with the community. In summary of our case study we make the following observations:

1. Contributions to both systems are larger than the average file sizes measured in Lines of Code, and are often implemented across multiple different files.
2. In both studied systems, accepted contributions are on average 3 times larger than rejected contributions, indicating a strong acceptance bias towards more substantial contributions.
3. From the quantitative part of our case study we observed that contributions often (over 50 % in ANDROID) propose changes that have already been put in place—either by other contributors or by the development team in the private code branches. Methods to raise awareness of contribution development efforts in order to prevent duplicated and thus wasted effort might provide an interesting subject for future research in contribution management systems.
4. The ratio of number of reviewers to contributors, as well as the number of contributions and frequency of contributions from individual contributors from the community are factors that impact both the time until a first feedback is given to contributors, and the time until a final decision is made on contributions

(as to accept or reject). We observed evidence in the ANDROID project that keeping both time spans low is an active effort, in order to keep contributors actively engaged.

5. The overall time of management for contributions plays an important role since contributors are available for feedback for only a limited time. If the contribution management process is too slow, feedback and requests addressed to the original contributors will be lost—and so is the potentially worthwhile contribution.

As a last observation, we want to note that contrary to past belief (Rigby et al. 2008), we have found that while successful contribution management can not only effectively deal with large contributions, both studied projects favour them over smaller contributions. We aim at investigating this observation in more detail in future research.

For future work, we aim to extend the conceptual model by studying additional open source projects, and their contribution management practices. In particular, we plan to extend our contribution management model with the pull-request process popularized by GIT providers such as GitHub and BitBucket. Furthermore, we plan to carry out an in-depth investigation of the key factors that influence attraction and retention of community members. Third, we plan to study the impact of the different decision practices in LINUX and ANDROID on project planning and feature integration.

Acknowledgements We would like to thank Richard Ellis for kindly providing us access to the LINUX kernel mailing list MBOX files. We would also like to thank the anonymous reviewers of the earlier version of this paper for their valuable feedback and comments.

References

Asundi J, Jayant R (2007) Patch review processes in open source software development communities: a comparative case study. In: HICSS '07: Proceedings of the 40th annual Hawaii international conference on system sciences. IEEE Computer Society, Washington, p. 166c. doi:10.1109/HICSS.2007.426

Bass FM (1969) A new product growth for model consumer durables. Manag Sci 15(5):215–227

Bettenburg N, Premraj R, Zimmermann T, Kim S (2008) Extracting structural information from bug reports. In: MSR '08: proceedings of the 2008 international working conference on mining software repositories. ACM, pp 27–30

Bettenburg N, Shihab E, Hassan AE (2009a) An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In: Proc. of the 25th IEEE intl. conf. on software maintenance (ICSM), Edmonton, Canada, pp 539–542

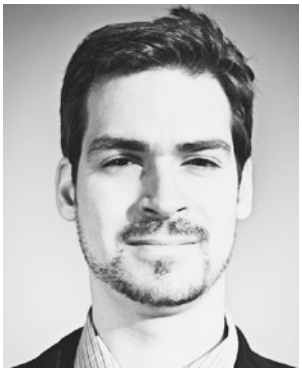
Bird C, Gourley A, Devanbu P (2007a) Detecting patch submission and acceptance in oss projects. In: MSR '07: proceedings of the fourth international workshop on mining software repositories. IEEE Computer Society, Washington, p 26. doi:10.1109/MSR.2007.6

Bird C, Gourley A, Devanbu P, Swaminathan A, Hsu G (2007b) Open borders? Immigration in open source projects. In: Proceedings of the fourth international workshop on mining software repositories, MSR '07. IEEE Computer Society, Washington, p 6. doi:10.1109/MSR.2007.23

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: MSR '09: proceedings of the 2009 6th IEEE international working conference on mining software repositories. IEEE Computer Society, Washington, pp 1–10. doi:10.1109/MSR.2009.5069475

Capiluppi A, Lago P, Morisio M (2003) Characteristics of open source projects. In: CSMR '03: proceedings of the seventh European conference on software maintenance and reengineering. IEEE Computer Society, Washington, p 317

Charmaz K (2006) Constructing grounded theory: a practical guide through qualitative analysis. Constructing grounded theory. SAGE Publications. http://books.google.ca/books?id=v1qP1KbXz1AC
Cleveland WS, Devlin SJ (1988) Locally weighted regression: an approach to regression analysis by local fitting. J Am Stat Assoc 83(403):596–610
Crowston K, Howison J (2005) The social structure of free and open source software development. First Monday 10(2)
Dobson AJ (2002) An introduction to generalized linear models, 2nd edn. Chapman and Hall/CRC, Boston
Duvall P, Matyas SM, Glover A (2007) Continuous integration: improving software quality and reducing risk (The Addison-Wesley signature series). Addison-Wesley Professional, Reading
German DM, Hassan AE (2009) License integration patterns: addressing license mismatches in component-based development. In: Proc. 31st int. conf. on soft. eng. ICSE, pp 188–198
Glaser BG, Strauss AL (1967) The discovery of grounded theory: strategies for qualitative research. Aldine
Hecker F (1999) Setting up shop: the business of open-source software. IEEE Softw 16(1):45–51
Hippel EV, Krogh GV (2003) Open source software and the "private-collective" innovation model: issues for organization science. Organ Sci 14(2):209–223. doi:10.1287/orsc.14.2.209.14992
Jiang Y, Adams B, German DM (2013) Will my patch make it? and how fast?—case study on the linux kernel. In: Proceedings of the 10th IEEE working conference on mining software repositories (MSR), San Francisco, CA, pp 101–110
Johnson PM (1998) Reengineering inspection. Commun ACM 41(2):49–52. doi:10.1145/269012.269020
Krishnamurthy S (2005) An analysis of open source business models. In: Perspectives on free and open source software (making sense of the Bazaar). MIT Press, Cambridge, pp 279–296
Mockus A, Fielding RT, Herbsleb J (2000) A case study of open source software development: the apache server. In: ICSE '00: proceedings of the 22nd international conference on software engineering. ACM, New York, pp 263–272. doi:10.1145/337180.337209
Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: apache and mozilla. ACM Trans Softw Eng Methodol 11(3):309–346. doi:10.1145/567793.567795
Porter A, Siy H, Mockus A, Votta L (1998) Understanding the sources of variation in software inspections. ACM Trans Softw Eng Methodol 7(1):41–79. doi:10.1145/268411.268421
Raymond ES (2001) The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary. O'Reilly & Associates, Inc., Sebastopol, CA
Reed DP (2001) The law of the pack. Harvard Business Review
Rice J (1995) Mathematical statistics and data analysis. Statistics series. Duxbury Press. http://books.google.ca/books?id=bIkQAQAAIAAJ
Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the apache server. In: ICSE '08: proceedings of the 30th international conference on software engineering. ACM, New York, pp 541–550. doi:10.1145/1368088.1368162
Rosenblatt M (1956) Remarks on some nonparametric estimates of a density function. Ann Math Stat 27(3):832–837
Sethanandha BD, Massey B, Jones W (2010) Managing open source contributions for software project sustainability. In: Proceedings of the 2010 Portland international conference on management of engineering & amp; technology (PICMET 2010), Bangkok, Thailand
Strauss A, Corbin J (1990) Basics of qualitative research: grounded theory procedures and techniques. Sage Publications
Weissgerber P, Neu D, Diehl S (2008) Small patches get in! In: MSR '08: proceedings of the 2008 international working conference on mining software repositories. ACM, pp 67–76. doi:10.1145/1370750.1370767
Wnuk K, Regnell B, Karlsson L (2009) What happened to our features? Visualization and understanding of scope change dynamics in a large-scale industrial setting. In: 17th IEEE international requirements engineering conference, 2009, RE'09. IEEE, pp 89–98
Yin R (2009) Case study research: design and methods. Applied social research methods. SAGE Publications. http://books.google.nl/books?id=FzawIAdilHkC



Nicolas Bettenburg is a PhD candidate at Queen's University (Canada) under the supervision of Dr. Ahmed E. Hassan. His research interests are in mining unstructured information from software repositories with a focus on relating developer communication and collaboration to software quality. In the past, he has co-organized various conference tracks and has been a co-organizer of the Mining Unstructured Data workshop since 2010.

Ahmed E. Hassan is the NSERC/BlackBerry Software Engineering Chair at the School of Computing in Queen's University. Dr. Hassan spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community. He serves on the editorial board of the IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and Springer Journal of Computing. Early tools and techniques developed by Dr. Hassan's team are already integrated into products used by millions of users worldwide. Dr. Hassan industrial experience includes helping architect the Blackberry wireless platform at RIM, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. Dr. Hassan is the named inventor of patents at several jurisdictions around the world including the United States, Europe, India, Canada, and Japan.

Bram Adams is an assistant professor at the École Polytechnique de Montréal, where he heads the MCIS lab on Maintenance, Construction and Intelligence of Software (http://mcis.polymtl.ca). He obtained his PhD at Ghent University (Belgium). His research interests include software release engineering in general, and software integration, software build systems, software modularity and software maintenance in particular. His work has been published at premier venues like ICSE, FSE, ASE, MSR and ICSM, as well as in major journals like TSE, EMSE, IST and JSS. Bram has beem program co-chair of the ERA-track at the 2013 IEEE International Conference on Software Maintenance (ICSM) and of the 2013 International Working Conference on Source Code Analysis and Manipulation (SCAM), and he was one of the organizers of the 1st International Workshop on Release Engineering (RELENG 2013), see http://releng.polymtl.ca.

Daniel M. German is assistant professor in the Department of Computer Science at the University of Victoria. His main areas of research are software evolution, open source software development and intellectual property.