What Happens When The Bazaar Grows

A comprehensive study on the contemporary Linux kernel development model

Melissa Shihfan Ribeiro Wen

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program: Computer Science
Advisor: Prof. Paulo Roberto Miranda Meirelles
Coadvisor: Prof. Fabio Kon

During this work, the author was supported by National Council for Scientific and Technological Development - Brazil (CNPq)

São Paulo
April 20th, 2021

What Happens When The Bazaar Grows

A comprehensive study on the contemporary Linux kernel development model

Melissa Shihfan Ribeiro Wen

This is the original version of the
thesis prepared by the candidate
Melissa Shihfan Ribeiro Wen, as
submitted to the Examining Committee.

Abstract

Melissa Shihfan Ribeiro Wen. **What Happens When The Bazaar Grows: *A comprehensive study on the contemporary Linux kernel development model***. Thesis (Masters). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

The popularity and consolidation of many Free/Libre Open-Source Software (FLOSS) projects in the information technology (IT) market keep industry and academia interested in identifying practices that can be beneficial to the software development process. Two decades ago, a set of practices observed in the Linux kernel development was used to characterize the FLOSS development model as a "noisy bazaar". However, since then, the FLOSS ecosystem diversified its forms of development. The Linux kernel project has also undergone notable transformations in its community and development processes toward professionalism and civility. FLOSS projects usually have a community supporting its development and organically producing plentiful information to describe how, when, and why a particular change occurred in the source code or the development flow. Although the existence of several studies on the FLOSS phenomenon and its development, these essential sources of information have been overlooked due to the informality and socio-technical challenges for data collection and analysis. Neglect of these resources may have led some studies to outdated and shallow results regarding FLOSS development practices. Bearing this in mind, we considered the great wealth of open-access materials and the Linux project relevance and protagonism on FLOSS phenomenon to mitigate the distance between what is investigated by academia and what is observed in practice on the development of the Linux kernel. We designed a multi-method investigation to cover academics' and practitioners' perspectives on the project's socio-technical aspects. We used a multivocal literature review, examining peer-reviewed papers and grey literature, to accurately map the Linux kernel development community's current characteristics. We included the participant observation on the development community as a third perspective to discuss our findings and nuances involved in community-based development. We also synthesized a set of research strategies to review FLOSS community publications. As a result, this research summarizes the state-of-the-art and state-of-the-practice of the Linux kernel's contemporary development model. As an adjoining outcome of this work, we present a combination of research methods that could boost and guide future FLOSS ecosystems research.

Keywords: Free Software. Open Source Software. FLOSS Ecosystems. Linux Kernel. Multivocal Literature Review. Grey Literature Review. Participant Observation

Resumo

Melissa Shihfan Ribeiro Wen. **O Que Acontece Quando o Bazar Cresce: *Um estudo abrangente sobre o atual modelo de desenvolvimento do kernel Linux***. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

A popularidade e consolidação de muitos projetos de Software Livre (FLOSS) no mercado de tecnologia da informação (TI) mantém a indústria e o meio acadêmico interessados em identificar práticas que podem ser benéficas para o processo de desenvolvimento de software. Duas décadas atrás, um conjunto de práticas observadas no desenvolvimento do kernel Linux foi usado para caracterizar o modelo de desenvolvimento FLOSS como um "bazar barulhento". Porém, desde então, o ecossistema FLOSS diversificou suas formas de desenvolvimento. O projeto do kernel Linux também passou por transformações notáveis tanto em sua comunidade quanto nos seus processos de desenvolvimento em direção ao profissionalismo e à civilidade. Os projetos FLOSS geralmente têm uma comunidade apoiando seu desenvolvimento e produzindo organicamente muitas informações para descrever como, quando e por que uma determinada mudança ocorreu no código-fonte ou no fluxo de desenvolvimento. Apesar da existência de diversos estudos sobre o fenômeno FLOSS e seu desenvolvimento, essas fontes essenciais de informação têm sido negligenciadas devido à informalidade e aos desafios sociotécnicos de coleta e análise dos dados. A negligência desses recursos pode ter levado alguns estudos a resultados desatualizados e superficiais em relação às práticas de desenvolvimento FLOSS. Tendo isso em mente, consideramos a grande riqueza de materiais de acesso aberto e a relevância e protagonismo do projeto Linux no fenômeno FLOSS para mitigar a distância entre o que é investigado pela academia e o que é observado na prática no desenvolvimento do kernel Linux. Projetamos uma investigação multi-método para cobrir as perspectivas de acadêmicos e profissionais sobre os aspectos sociotécnicos do projeto. Usamos uma revisão de literatura multivocal, examinando artigos revisados por pares e literatura cinzenta, para mapear com precisão as características atuais da comunidade de desenvolvimento do kernel Linux. Incluímos a observação participante no na comunidade de desenvolvimento como uma terceira perspectiva na discussão das nossas descobertas e das nuances envolvidas no desenvolvimento baseado na comunidade. Também sintetizamos um conjunto de estratégias de pesquisa para revisar as publicações da comunidade FLOSS. Como resultado, esta pesquisa resume o estado da arte e o estado da prática do modelo de desenvolvimento contemporâneo do kernel Linux. Como resultado adjacente deste trabalho, apresentamos uma combinação de métodos de pesquisa que podem impulsionar e orientar futuras pesquisas de ecossistemas FLOSS.

Palavras-chave: Software Livre. Ecossistemas de Software Livre. Kernel Linux. Revisão da Literatura Multivocal. Revisão da Literatura Cinzenta. Observação Participante

List of Abbreviations

FLOSS	Free/Libre Open-Source Software
SE	Software Engineering
MLR	Multivocal Literature Review
GLR	Grey Literature Review
SLR	Systematic Literature Review
IIO	Industrial I/O - Linux Kernel Subsystem
DRM	Direct Rendering Manager - Linux Kernel Subsystem
USP	University of São Paulo, Brazil
IME	Institute of Mathematics and Statistics, University of São Paulo
FLUSP	FLOSS at USP - Student Special Interest Group
CCSL	FLOSS Competence Center, Brazil

List of Figures

List of Tables

Contents

Appendices

Annexes

Chapter 1

Introduction

Free/Libre Open Source Software (FLOSS) development has been the subject of investigation in academia and industry. Their interest stems from the popularity of various software projects and their successful use of peculiar management and training practices to conduct a high-quality collaborative, geographically distributed, community-based software development effort.

The search for effective and efficient ways of producing computer programs is at the heart of the Software Engineering (ES) discipline. Nevertheless, a certain detachment from the practice observed throughout the history of academic studies on FLOSS may lead many investigations to carry myths and superficial understanding of its phenomenon and an outdated view of the development model of one of the largest, most famous, and most consolidated projects in the history of FLOSS phenomenon, the Linux kernel.

The Linux kernel development is connected in many ways to the history of the free software and open-source movements. Also, few FLOSS projects have managed to survive three decades and still have a pulsating development as the Linux kernel. Its characteristics inspired software development models. Its long history, large community, and the interest of software engineering studies in understanding its practices provide a great wealth of open-access sources of information available for investigation.

The primary objective of this thesis is to describe the current development community of the Linux kernel project, considering community publications and academic studies. Our purpose is to capture community attributes, challenges, and concerns presented by practitioners and compare their coverage by ES investigation. We also aim to identify research techniques and data sources able to mitigate the distance between what is investigated by academia and what is observed in practice on the development of the Linux kernel – the project that inspired one of the first initiatives to define a FLOSS development model, the essay "The Cathedral and the Bazaar" (Raymond, 1999). We[1] consider that the real-world gaps in many studies on this kind of software are not caused by a lack of methodological discipline and rigor in an investigation but rather by the absence of

[1] I often use "we" to describe the results of the collaborative process that involves my advisors and/or other researchers at the FLOSS Competence Center of the University of São Paulo (CCSL); the use of "I" is specific to work performed individually.

research approaches that use methods to shed light on the practice and endorse scientific findings.

In short, this investigation proposes to extract the attributes used to characterize the Linux kernel development community from the state-of-the-art and state-of-the-practice. We examine the outputs from academia and the Linux kernel community publications to capture the attributes used to describe the Linux development community model, including project management characteristics, organizational structure, workflow, and decision-making process. We also participated in the daily community to explore other nuances of the project characteristics, making industry-as-laboratory. From this mapping, we deliver a comprehensive study on the Linux kernel project's development community in light of the social and behavioral aspects of a distributed, community-based workflow.

1.1 Problem outline

At the end of the nineties, Raymond (1999) described two supposedly antagonistic FLOSS development models through an essay named "The Cathedral and The Bazaar". These models were based on his observations of the Linux kernel[2] development and lessons learned during his own software project, the *fetchmail*. The seminal essay aimed to disseminate an apolitical and unbiased term to define the development model of free software projects, the term *open-source*. Consequently, a new wave inside the free software communities emerged and brought along with it firms interested in the benefits of these open-source practices that enable innovation, technical support from an external community, and "many-eyes" assisting the software development (Crowston et al., 2012).

Companies' participation in FLOSS development projects has caused changes in their communities, development processes, and business strategies. As a result, these elements lost several free software characteristics and became more mainstream and viable (Fitzgerald, 2006). The development of many FLOSS projects is now supported by large companies, either directly or through contributions from paid employees (Lakhani and Wolf, 2003). Thus, FLOSS projects are less and less bazaar-like as strategic planning becomes essential. At large, it became a public product produced by the private initiative (Crowston et al., 2012). Other transformations came from the project evolution in terms of structure and technologies for supporting distributed management and development, such as version control systems, instant messengers, and authentication keys. In short, FLOSS development has followed the web-based pace of globalization and innovation.

In the third decade of existence, the Linux kernel project also has transformed itself. It steadily grows in terms of source-code size, developer team size, and variety of markets. With the popularity of Android smartphones, the kernel is used by a large portion of the global population. Its current development model is in a different scenario and presents other aspects of those on which the Bazaar model was based. A core group of developers takes most of the reviews and is also exclusively responsible for approving code changes, passed by a chain-of-trust to Linus Torvalds' official repository. Therefore, the risk of code instability and intractable bug insertion on the mainline may not vary with the number of testers and developers. This aspect resembles the Cathedral model's design and contrasts

[2] https://www.linuxfoundation.org/projects/linux

with the so-called "Linus' Law" defined by Raymond as "Given enough eyeballs, all bugs are shallow".

Many Software Engineering studies already dissociate the Linux kernel development from the bazar-like model. For Rigby et al. (2014), Linux is currently managed by a dictatorship where its review process has an average of two reviewers, and 44% of patches sent to the project are ignored and never merged to the code. However, for Fogel (2017), code "forkability", present in the Linux project, is why there are no real dictators in FLOSS projects, but a particular type of dictatorship, the "benevolent dictator" model. In this model, the dictator does not have a strict hold on the project, letting things work themselves out and usually making final decisions only if there is no consensus. For Lindberg et al. (2014), the Linux kernel follows a cathedral model, but Shaikh and Henfridsson (2017) argue that four governance types coexist in the Linux kernel development: autocracy, oligarchy, federation, meritocracy. This unclear definition reveals that understanding the Linux kernel development is an ongoing issue in academic investigations.

The annual increase in the number of publications of empirical work on FLOSS demonstrates the growing interest in the subject (Crowston et al., 2012). Notwithstanding the apparent consonance between industry and academia on the absorption of FLOSS practices or products in software development projects, Osterlie and Jaccheri (2007) reveal a lack of understanding of the FLOSS phenomenon in academic works, exposing the homogeneous and biased point of view of many studies and their findings. For the authors, the distancing of software engineering studies from the practice gives a chance to reproduce imprecision. Another aggravating factor of this imprecision is the initial and non-comprehensive use of non-formal materials, such as blog posts, videos, whitepapers, and websites. These materials are commonly produced by FLOSS practitioners in their day-to-day activities and are defined by Scacchi and Walt (2007) as *Software Informalism*.

Many studies look at some artifacts produced by the community. In general, they examine lines of code, commits, and emails. However, although Software Engineering publications widely use the term community-based to describe the FLOSS development model, academic works do not always examine what the community has said about the project of which it is a part. These mismatches led many software engineering studies to ignore the diversification of development forms in the FLOSS ecosystem and transformations on the Linux kernel development model after two decades since "The Cathedral and The Bazaar".

Given this scenario, for this thesis, we delimited the scope of some terms as follows:

- *Free Software*: refers to the Free Software movement, formalized by Richard Stallman in 1983
- *Open Source*: refers to the Open-Source movement, a movement to pitch a concept of "free software" development viable in the business world, as initially explicited by the Cathedral and Bazaar essay.
- *FLOSS, an acronym for Free / Libre and Open Source Software*: refers to both free and open software ecosystems. This term will be extensively used in discussions of practices and methods used by developers in the freely licensed software community.

- *FLOSS Phenomenon*: when the topic in discussion involves characteristics of the Free Software and Open-Source movements
- *FLOSS Development*: when the subject is limited to the workflow and set of techniques used in the production of this kind of software.

1.2 Research Questions

The Linux kernel project has a protagonism defining what we currently know as the bazaar-like FLOSS development model. Besides, academia and industry have shown the relevance of using FLOSS development practices for innovation and software quality. Accordingly, this research primary goal is to map the attributes that describe the contemporary Linux development community model considering the state-of-the-art and -practice of FLOSS development characteristics. We collect and analyze data from FLOSS community publication and Software Engineering studies on Linux kernel development, comparing our findings. I conduct a participant-observation in the Linux kernel community to reinforce convergences and address possible divergences between what is claimed by academia and observed in daily work practice.

The following research questions will guide our investigation:

RQ1. How do software engineering studies and Linux community publications describe the current Linux development community model?

RQ1.1. What attributes practitioners use to characterize the Linux development community? What are the current social and organizational challenges from the community perspective?

RQ1.2. Do Software Engineering studies already cover these topics?

RQ2. What research techniques can be used to examine a FLOSS project through its community publications?

RQ3. What are the possible gaps and opportunities for academic research in FLOSS development topics?

The Linux project's selection takes into account the role played by this project in the Free Software and Open-source movements. Also, much of the FLOSS model's academic understanding is based on assertions about the Linux project development model – or Bazaar model. Describing the contemporary characteristics of the Linux project's workflow and community, in contrast with its traditional characterization, will produce an updated view of the roles, rules, and restrictions present in the Linux development model, one of the major projects in FLOSS history.

1.3 Research Design

A variety of research methods supports FLOSS ecosystem studies, and the case study is, without doubt, the most common method. On the other hand, multi-method investigations are not largely used and are often designed to incorporate interviews into case studies,

surveys, and field studies (Crowston et al., 2012). Researchers are interested in elucidating factors and features of FLOSS development that increase the chances of a software project succeeding. According to Fogel (2017), it is not difficult for a FLOSS project to achieve technical success; however, after initial success, it needs a social foundation and a robust developer base to handle the growth or loss of skilled workforce.

Over the last two decades, the Linux kernel project grew in size, diversity, and maturity. Besides examining the human interaction networks intrinsic of community-based development, this study intends to benefit from the wealth of information present in the non-academic publications. We aim to expand the Linux kernel ecosystem's academic understanding by providing a socio-technical picture of the contemporary Linux kernel development community.

We conducted a multivocal literature review (Vahid Garousi et al., 2018) to describe the Linux development community model from the academia and community perspectives. When reviewing multivocal literature, we examine documents from both grey literature and academic studies on FLOSS development. To the best of our knowledge, no single study has focused on identifying FLOSS research opportunities in the Linux kernel development model by comparing characteristics mapped in Software Engineering studies with those presented in FLOSS community publications. Moreover, conducting a multivocal literature review in the Linux kernel project reveals mismatches and misunderstandings between academia and industry communities about one of the most prominent and valuable projects for the FLOSS phenomenon and history.

We also collected and analyzed qualitative data from my participant-observation in the Linux kernel project community. We verified findings from this ethnographic method with those obtained in the multivocal literature review to discuss a potential misleading characterization of the Linux kernel development model and Software Engineering research opportunities in FLOSS.

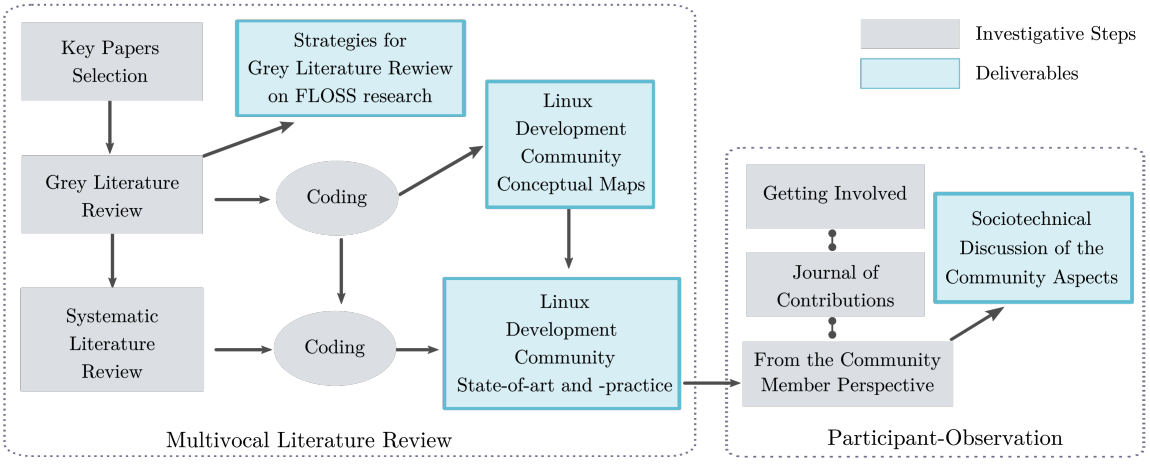


Figure 1.1: *Practical and Theoretical Study Workflow*

Figure1.1 summarizes the research strategies, methods, expected outcomes, and the current stage of the investigation. This research is divided into four phases: the first phase of the acquaintance of data sources and research methods; the second phase of multivocal

literature review; the third phase of the ethnographic case study; and the fourth and final phase of confirming results by triangulation. Each phase is detailed below.

Phase I is a warm-up on the studies of the FLOSS phenomenon and its development model. At this stage, we conducted a preliminary analysis of traditional and non-traditional literature regarding elements commonly used to describe FLOSS development models and community workflows. These introductory materials support the construction of a search string for the following steps. Another initial step is the introduction of the observer in the Linux kernel community. In this phase, I had to sharpen my code skills and started dialogues with community members to collect data by participant observation. Due to the steep learning curve for becoming a kernel developer and having some project participants in our research group, we chose to start this preparatory phase of participant observation as soon as possible.

Phase II summarizes the state-of-the-art and -practice via a multivocal literature review. The grey literature review identifies attributes commonly used by the FLOSS community to describe the Linux kernel development community. The systematic review of traditional literature and, finally, the combination of findings enables creating a comparative mind map of the contemporary Linux kernel development community from academic researchers' and practitioners' perspectives. Examining one data source has the inherently potential to bring bias when reviewing another one. Considering this risk, we started the multivocal review by grey literature. We prioritize the grey literature review because it is still poorly systematized on FLOSS research, but these materials are resources closer to practitioners' routines, gathering diverse undocumented information and decisions.

Phase III strengthens Phase II's findings through an in-depth ethnographic case study on the Linux kernel community. At this stage, we focus on understanding people, project culture, and their related social and work practices to shed light on mismatched understandings between academic researchers and practitioners. I participate in the community's daily life to capture undocumented characteristics of the Linux kernel development. As an observer, I also identify misleading information and shallow approaches for some subjects of software engineering studies that become research opportunities on FLOSS.

Thus, we design an ethnographic case study of the Linux kernel project. Data collection follows a qualitative method of participant-observation, where an observer becomes a member of the community under investigation, participating in daily tasks and different situations for members' interaction. On the one hand, I volunteered in FLUSP[3], a group focused on FLOSS project development at the University of São Paulo. I performed necessary activities of communication and contribution to participate in a Linux kernel subsystem's development process, the Industrial I/O[4]. In the continuity of the investigation, I joined another Linux subsystem, the Linux DRM for GPU drivers[5], taking a longer participant-observation that included the performance of different community roles: a volunteer newcomer; an independent developer for the Google Summer of Code program[6];

[3] https://flusp.ime.usp.br/
[4] https://www.kernel.org/doc/html/latest/driver-api/iio/index.html
[5] https://www.kernel.org/doc/html/latest/gpu/introduction.html
[6] https://summerofcode.withgoogle.com/

co-maintainer of a subsystem driver; and co-mentor of a project for Outreachy[7], a paid internship program. Driver maintainership involves clerical and development tasks, such as reviewing contributions to the driver, keeping driver development code up-to-date, bug-fixing and synchronizing driver code to subsystem updates. Mentorship also involves many activities, such as evaluating candidates, designing the internship projects, guide newcomers through drivers and subsystem code, explaining concepts, introducing them to the subsystem community, review contributions. Therefore, this multi-role experience also enables virtual interactions with other community members to comprehend their behaviors, communication styles, and development process perceptions with mine.

Finally, **Phase IV** triangulates the findings from the GLR, SRL and participant and non-participant observation in the case of the Linux project. Also, we discuss the advantages, challenges and limitations of this multi-method research design to help future research in the choice of appropriate research methods.

1.4 Claimed Contributions

This thesis aims to investigate features of the Linux kernel community in an up-to-date view of the Linux kernel development model. From this goal, we claim that this work delivers four primary original contributions:

C1. A comprehensive characterization of the contemporary Linux kernel development community. The systematic grey literature review and content analysis resulted in a mind map of main concepts used by Linux kernel community members to describe the development of its common-source project. We grouped the concepts in main subjects regarding characteristics of the community as a unit, members concerns and natural and legal persons participation. **C1.1 A map of potentiallly misleading information and gaps between theory and practice in the contemporary Linux kernel development community.** We design a multi-method investigation to map convergencies in theory and practice on the Linux kernel development community. We combine and compare findings from reviewing software engineering studies and community publications to provide an up-to-date description of the project development community. We participated in the daily life of the community to observe undocumented features. Consequently, we produce a comprehensive and critical mapping of misleading information and gaps in the Linux kernel development model's academic understanding. Filling these gaps improves the investigative design and increases the relevance of future FLOSS research.

C2. Community characteristics and social-technical nuances that shapes a FLOSS project development. Many FLOSS projects are supported by a community of developers. In the Linux kernel project, the development model is defined by concepts that describe its release process, artifacts, software-product, and community. Discussion on the Linux kernel structures, rules, and roles used to organize a community of contributors is dispersed in Software Engineering studies. We thus provide a rationale in which community aspects set the pace of the current Linux kernel development and are the basis of the sustainability of the project. Therefore, we argue that FLOSS development should be evaluated from a social perspective, where social aspects impact development processes,

[7] https://www.outreachy.org/

growth, and continuation of a project. We also explored the advantages of participant observation to analyze the content and discuss the multivocal review results in one of the most successful projects in the FLOSS phenomenon.

C3. Guidelines to examine FLOSS projects through its community publications. We systematize the entire process of reviewing and analyzing content from grey literature produced by the Linux kernel community. The methods, challenges, and adaptations reported here can guide future grey literature review on FLOSS and Software Engineering studies. From our lessons learned in this process, FLOSS Researchers can also anticipate issues and design a protocol to prevent bottlenecks when dealing with a large amount of content, usually poorly structured.

C4. A combination of research strategies that could boost research on FLOSS ecosystems. We resort to different systematic literature reviews and qualitative methods to summarize state-of-art and -practices in the Linux kernel development model. The multivocal literature review is a powerful method to include negative results, current discussions, and emerging research topics in software engineering and the software industry. It also captures practitioners' point of view that, combined with a participant-observation, provides a comprehensive understanding of FLOSS project characteristics and transversal social aspects of FLOSS development. We use data from community publications, software engineering studies, and participant-observation to expand our research on the Linux kernel ecosystem with concepts from theory, practice, and development trenches. To date, no other Software Engineering studies on FLOSS have systematically reviewed community publications from FLOSS projects and, moreover, combined findings with a systematic literature review.

C1 and C1.1 resulted from answering RQ1 and RQ1.2. C2 is a consequence of answering RQ1.1. C3 is related to the RQ2. Finally, RQ3 covers C4 and C1.1.

1.4.1 Publications in the area of FLOSS Development

We have published a total of five conference and journal papers based on this research or covering diverse themes related to topics of this thesis.

P1. Wen, M., Leite, L., Kon, F., Meirelles, P. (2020). Understanding FLOSS through community publications: strategies for grey literature review. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2020.

P2. Wen, M., Siqueira, R., Lago, N., Camarinha, D., Terceiro, A., Kon, F., Meirelles, P. (2020). Leading Successful Government-Academia Collaboration Using FLOSS and Agile Values. Journal of Systems and Software. Volume 164.

P3. Wen, M., Meirelles, P., Siqueira, R., Kon, F. (2018). FLOSS project management in government-academia collaboration. Open Source Systems: Enterprise Software and Solutions, OSS 2018. (Best Paper Award)

P4. Siqueira, R., Camarinha, D., Wen, M., Meirelles, P., Kon, F. (2018). Continuous delivery: Building trust in a large-scale, complex government organization. IEEE Software, 35(2).

P5. Meirelles, P., Wen, M., Terceiro, A., Siqueira, R., Kanashiro, L., Neri, H. (2017). Brazilian public software portal: An integrated platform for collaborative development. Proceedings of the 13th International Symposium on Open Collaboration, OpenSym 2017.

1.5 Thesis Structure

This thesis consists of five more chapters. Chapter 2 presents a history of the FLOSS phenomenon – history, movement, and related development model – and the Linux kernel both as an operating system kernel and a FLOSS project. Chapter 3 describes the research methods selected to guide this study, discussing advantages, challenges, and specificities. Chapter 4 proposes strategies for conducting Grey Literature Reviews on FLOSS based on guidelines from related works on Software Engineering literature review. We present the decisions taken and adaptations made to examine and analyze FLOSS community publications with different formality levels. This process resulted in a comprehensive mind map describing the Linux kernel development community. We also discuss the community concerns regarding process scalability, member behavior, and project subsistence from the practitioner's perspective. Chapter 5 presents related works on the Linux development model characteristics as systematic literature review findings. We compare results here to those from the grey literature review. We include a third point of view from a participant-observation to discuss the characteristics and concerns of the Linux kernel community. We also map and discuss the matches and mismatches of the attributes considering the theory and practice of Linux kernel development. Finally, Chapter 6 concludes this research. We answer our research question and discuss opportunities for future Software Engineering research on FLOSS from the gaps mapped by our multivocal literature review.

Chapter 2

Background

2.1 Free/Libre Open Source Software

The Free Software and Open Source movements have in their bases the opposition to established policies and models of the software development industry of their times. While the Free Software movement was in opposition to the proprietary software industry's dominance, the Open Source advocated the superiority of a bazaar-like development over the hierarchical model typically used in proprietary software projects. After more than 30 years of the Free Software movement and 20 years of Open Source, academia lacks a complete understanding of FLOSS development's possible models. Hence, revisiting the history of Free and Open Source Software is essential to uncover loose ends in academic investigations.

Free Software Movement

The first steps of the Free Software movement were given in 1983 when Richard Stallman announced the GNU project. The project's goal was to build a Unix-compatible operating system to be made available for free (Stallman, 1983; Stallman, 2018). In Stallman's view, the existence of a free operating system would strengthen the cooperative development of computing communities (F. S. Foundation, 2017) that had been weakened by the dominance of the private software industry. This movement was consolidated two years later with the publication of the GNU Manifesto in Dr. Dobb's Journal of Software Tools and the Free Software Foundation (FSF) creation. FSF supports the GNU project and aims to promote the development and use of free software and ensure the freedom to copy, study, modify, and redistribute software. In 1989, the same project created the "GNU General Public License" (GPL) (F. S. Foundation, 2007), a copyleft scheme that eliminates usage restrictions in software and other kinds of intellectual production.

Only in 1992, with the Linux kernel incorporation, the GNU project became a complete operating system. The existence of a free operating system and the GNU/Linux project's success led to the growth of software projects under the GPL license and its volunteer communities, which, in turn, led to increased commercial interest in its development process.

Open Source Movement

In 1997, through a set of observations made in the Linux kernel community and lessons learned from his experiences with other software projects, Eric Raymond presented the essay "The Cathedral and the Bazaar". In this essay, Raymond (1999) defines two antagonistic software development models: the cathedral model and the bazaar model. Projects classified as cathedrals would be those projects that follow a hierarchical model of development, with long periods between the release of new versions and the users' little participation in the software construction process. In opposition to that, Raymond (1999) claims that the bazaar model, like Linux, is open to receiving any user's contribution, with quick releases and assuming the risk of failures and bugs. For him, the bazaar model was like a horde of anarchists competing and surpassing the hierarchical form of closed software development.

Th term Open Source was used by Raymond to propagate a depoliticized and ideologically unbiased concept of free software projects. Raymond defined that the bazaar development model, or Linux model, would be the model to be followed by open-source software. With an initial set of Linux development characteristics and an apparent dissociation of moral issues, adopting a bazaar model intensified commercial actors' participation in Open Source projects. *"Release early and often. Delegate everything you can. Be open to the point of promiscuity"* was a premise that won the developers' sympathy and users of the projects and companies. They could see advantages in innovation and improvement of their products by opening the code for contributions.

Bruce Perens edited the Debian Free Software Guidelines to form the Open Source Definition and registered a certification mark on the term Open Source. He transferred the mark's ownership to Raymond, and they together formed the Open Source Initiative, *"an organization exclusively for managing the Open Source campaign and its certification mark"* (DiBona and Ockman, 1999).

Raymond's essay has become a landmark for the free software universe until now. While on the one hand, the essay sowed the investigation, understanding, and implementation of good practices of OSS development. On the other hand, the defense of the term open source as an apolitical synonym of free software disturbed some practitioners and generated a rupture between them. Stallman published an online article stating that Open Source software misses the point of Free software (Stallman, 2019). Perens stated the concept of Free Software was outdated when he opted for Open Source. However, some years later, he explained that Open Source was coined to promote the concept of Free Software to business people and regrets that the Open Source Initiative deprecated Richard Stallman and Free Software. For him, *"Open Source licenses and Free Software licenses are effectively the same thing."*[1]

2.1.1 FLOSS Development Model

According to Mockus et al. (2002), FLOSS processes can produce high-quality and widely deployed software; however, the exact means responsible for the success are frequently debated. Meritocracy, cooperative spirit, and code transparency are all part of

[1] https://perens.com/2017/09/26/on-usage-of-the-phrase-open-source/

this, but they are not enough to explain a project's daily routine and how they resolve conflicts (Fogel, 2017).

We can consider the Cathedral and Bazaar models presented by Raymond, 1999 as a preliminary initiative to characterize freely licensed software development models. When observing free software projects of that time, Raymond separated in two sets the characteristics found by him in a personal project and projects like Linux and GCC, as summarized below:

The Cathedral

- **Development team:** a group of few people, well-trained and working in isolation.
- **Release process:** does not include beta versions. Do not release a version of the software until it has as few bugs as possible so as not to undermine the user's patience. Uses very long release intervals which increase the user expectations for a perfect version.
- **Debugging:** takes a long time. A bug is a complicated and profound phenomenon. Only a small group of developers can solve them.

The Bazaar

- **Development team:** composed of co-developers available via the internet. Contributors having different schedules and approaches, motivated by ego satisfaction, perception of constant improvements of the software coming from their works, personal needs, or love. Users can also be co-developers, helping to find, suggesting fixes, and improving code faster. Contributions are made by people who are sufficiently interested in using the software, learning how it works, trying to find solutions to the problems encountered, and actually producing a reasonable fix. The coordinator needs to be able to communicate well and attract good people. The developers are selfish agents who try to maximize their productivity and, during this process, spontaneously auto-organizing the development workflow.
- **Release process:** Make software versions available as soon as possible and as often as possible. Perfection is achieved when the code produced is not only efficient but also as simple as possible. Coding, enhancement, and debugging is parallelizable when the mode of development is based on rapid interactions.
- **Debugging:** Maximize the number of person-hours debugging and developing even if it costs code instability and introduction of an intractable bug. Given a sufficient number of testers and developers, almost all problems will be characterized quickly (by someone), and the solution will be evident to others. A bug is a superficial phenomenon (easily visualized), or is brought to the surface quickly when a version is available to several observers. Debugging is parallelizable and does not require much coordination, just a useful reference. In practice, debug rework (duplication) is rarely a problem, and this rework is reduced with increasing launch frequency. More users find more bugs because they can stress software in different ways. There is always the option to use a previous (stable) version if a critical bug is found in the current tree (unstable).

According to Raymond (1999), starting a project already with the shape of a bazaar is not trivial since someone needs a plausible promise to create a community, i.e., something executable and testable for this community. Raymond (1999) states that more important than being brilliant, a project coordinator should be able to identify promising ideas from other people's projects. Coordinators should know when to restrict the innovations and complexities in favor of code robustness and objectivity. Moreover, they should understand that the free software community values its reputation. Therefore, they should pressure people not to initiate development efforts that are not worth moving forward.

After more than two decades, the FLOSS development based on geographically distributed communities has evolved in an OSS 2.0 due to the growing participation of commercial actors (Fitzgerald, 2006). The author characterizes both FLOSS in its origin as its current version and points out possible gaps between the focus of OSS research and the OSS phenomenon itself. Also, investigations that mine data from the contributions' flow of different projects point to the FLOSS dissociation of the bazaar model.

Fogel (2017) states that as a project gets old, it tends to move from a benevolent dictatorship model to an openly democratic system with group-based governance. This model would then be a "consensus-based democracy", where the group works in consensus most of the time, and there is a formal voting mechanism. Finally, after metamorphosing into a group-based system, a project rarely moves back.

Literature reviews supplemented and updated this new shape of FLOSS development. Scacchi, Feller, et al. (2006) reviews empirical studies of FLOSS projects to identify practices, processes, dynamics, and other socio-technical concerns involved in these ecosystems and research opportunities in this field of study. The author also points to the relevance of resources and capabilities supporting this kind of development. Osterlie and Jaccheri (2007) conduct a critical review of the formal literature to investigate, through discourse analysis, how software engineering research has treated *open-source* as a homogeneous phenomenon (Osterlie and Jaccheri, 2007). This analysis demonstrates various points where academia has built biased concepts about the practice and advocates for the heterogeneity and multidisciplinarity of *open-source* development (OSSD).

Kon et al. (2011) discuss how the wealth of information available in FLOSS project artifacts can benefit Software Engineering research and education. They also evaluated where the Brazilian Software Engineering research community stands with regard to FLOSS. Crowston et al. (2012) reviewed commercial publications between 1999 and 2006 related to empirical research in the development of FLOSS. The purpose of the paper is to synthesize what academia has known and unknown about the process and public/private practices involved in the FLOSS development. Ultimately, Steinmacher et al. (2015) selected 20 studies that provide empirical evidence of newcomers' barriers when contributing to a FLOSS project. They classified 15 barriers into five categories and also problems according to their three origins.

Considering the diversity of FLOSS projects, Capiluppi, Lago, et al. (2003) examined almost 400 projects to find FLOSS project properties. Their work extracted generic characterization (project size, age, license, and programming language), analyzed the average number of people involved in the project (developers, subscribers, and core team), the community of users, modularity, and documentation characteristics.

Some FLOSS practices overlap with agile values regarding individuals and interactions, working software, customer collaboration, responsiveness to change, and software project management (Beck et al., 2010; Javdani Gandomani et al., 2013). Besides, FLOSS projects rely on self-organized teams and team-wide shared and coherent goals (P. Adams and Capiluppi, 2009; Tsirakidis et al., 2009). Warsta and Abrahamsson (2003) found differences and similarities between agile development and FLOSS practices. Open communication, project modularity, the users' community, and fast response to problems are just a few of the FLOSS ecosystem practices (Capiluppi, Lago, et al., 2003; Warsta and Abrahamsson, 2003). The authors argued that FLOSS development might differ from agile in their philosophical and economic perspectives; on the other hand, both approaches share similar work definitions.

We have also reported the reproducibility of sound FLOSS development practices in a software development collaboration (Wen, Siqueira, et al., 2020). We explained FLOSS community development standards, such as developers being also users of the system under development. FLOSS project is usually divided into fronts of self-organized development teams, identifying among developers the leadership roles. They use open means of communication to code review, track activities, and organically document the project, processes, and technical decisions. Finally, project stakeholders participate in the development routines and workflow through their employees.

2.2 The Linux Kernel

Linux is a monolithic kernel of the family of Unix-like operating systems. It is not a complete Unix operating system since it does not include filesystem utilities, windowing systems, desktop environment, text editors, and compilers. Unlike Unix, the Linux kernel is under the GNU General Public License version 2 only (GPL-2.0), with an explicit syscall exception[2]. This license protects the code of commercial ownership and benefits it with developing the userspace project GNU. Also, the project is open and available to anyone to study (Bovet and Cesati, 2005).

According to Tanenbaum, 2014, Linux is a rewrite of MINIX, with numerous ideas from this microkernel project and other new features. The Linux kernel uses modules to achieve microkernel advantages. A module is an object file whose code is linked to the kernel at runtime. This object code usually implements features at the kernel's upper layer (Bovet and Cesati, 2005).

The Linux kernel is divided into three levels, as shown in Figure 2.1: the top-level implements basic functionalities of the System Call Interface, such as read and write; the architecture-independent kernel code is in the middle, and the architecture-dependent code is in the bottom-level (Jones, 2007).

2.2.1 The Linux Project

The Linux kernel project currently has almost 25 million lines of code and much more than 1,500 active developers from more than 250 different companies (or from no company).

[2] https://www.kernel.org/doc/html/latest/process/license-rules.html

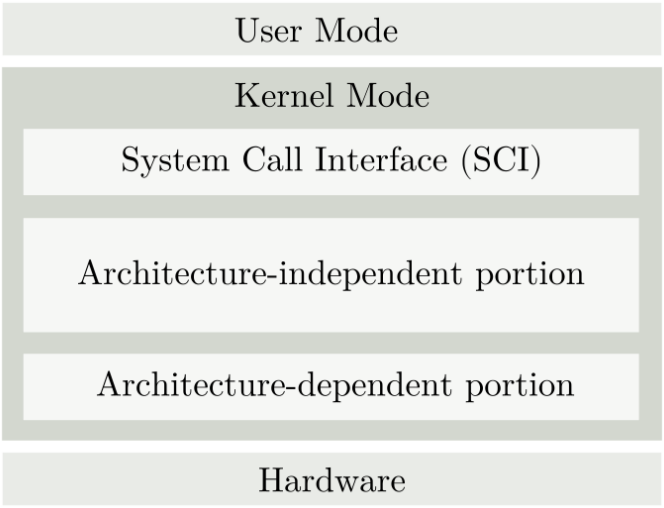


Figure 2.1: *An Overview of the Linux Architecture*

Its development process differs significantly from proprietary development methods, with a community-based scheme, a patch-oriented development process, and a time-based release process.

Linux History

Linux is an operating system kernel that began to be written by Linus Torvalds in 1991 (Torvalds, 1992). It was initially private, but in 1992 Linus made it free software, enabling the GNU project to adopt this kernel and thus creating the GNU/Linux operating system. The first stable version of Linux (1.0) was released in 1994, with about 165,000 lines of code (Tanenbaum, 2014). Due to the full participation of the GNU community - concerned with the compatibility of the system with the stable version of the kernel - and the dynamics of geographically distributed volunteers coordinated by Torvalds, the project grew to such an extent that, five years later, version 2.2.0 already had 1.8 million lines of code (Jones, 2007).

Throughout its three decades, the Linux creator, Linus Torvalds, has repositioned the project regarding the Free Software and Open-Souce movements. In 2007, Torvalds positioned itself against the adoption of GPLv3 (Torvalds, 2007), created by the Free Software Foundation. More recently, in a TED of 2016, he stated that he was never really concerned with following an open source methodology or free software policies (McManus, 2016). He just opened the code for feedback and created methods and tools (such as Git) that made it possible to shape development in the most comfortable possible way for him. Finally, in a letter recently sent to the kernel project mailing list in 2018, Torvalds licenses himself for a brief period from the coordination of Linux due to strong disagreements with the community regarding his way of conducting the project (Torvalds, 2018). Greg Kroah-Hartman took over Torvalds' coordination responsibilities and, after a month, Torvalds returned to Linux development (Statt, 2018). At that time, the controversial code of conflicts was replaced by a new code of conduct as a symbol of a welcoming and inclusive community that cares about its developers.

The Project Structure

The kernel code is organized as a set of subsystems[3] such as process scheduler, memory management, device driver infrastructure, networking, and filesystems (Community, 2017) (depicted by Figure 2.2). Usually, each subsystem has designated maintainers. A maintainer is the project role responsible for managing and accepting contributions into a specific subsystem code repository. The kernel project uses the Git source management tool to coordinate contributions. Contributions are formatted and sent to the Linux kernel project, as a patch. A patch is a text document describing the differences between two different versions of a source tree. These contributions (or patches) are grouped according to a particular need or interest. This collection of patches are arranged in a tree. And each tree works as a place of a specific development.

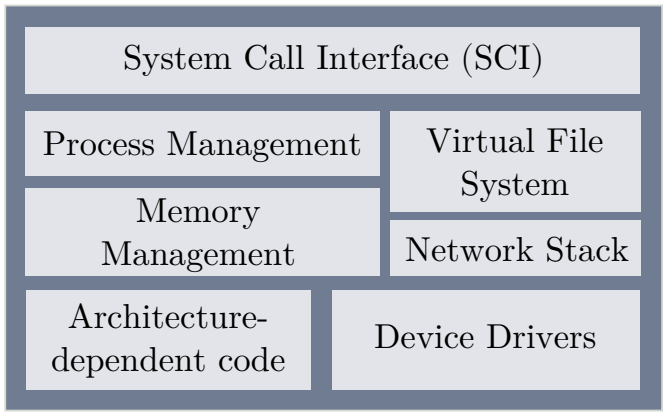


Figure 2.2: *Linux Subsystems*

A contribution sent to a subsystem does not go directly to the mainline tree of Linux. Each subsystem maintainer has a version of the kernel source tree, which enables the maintainer to track patches. The chain of subsystem trees guides the flow of patches to Torvalds' source tree (Corbet, 2008).

At the beginning of a development cycle, a "window" is opened to merge the code of changes accepted by the maintainers of a given subsystem into the mainline kernel. Changes in each subsystem trees are merged to Torvalds' source tree via pull requests. After this time, Torvalds declares the window as closed and releases a first intermediate release (Documentation, 2019). There is a somewhat informal process to ensure that a change is in the quality and relevance appropriate for the official repository. This evaluation is taken by top-level maintainers and also by Torvalds. Sometimes, he takes special attention to some particular patch; however, in general, he trusts maintainers will not send bad code.

Linux development processes extensively rely on human evaluation and subjectivity. Notwithstanding the building of a chain-of-trust, the community has raised questions

[3] https://lwn.net/Articles/844539/

about Torvalds' behavior (Torvalds, 2018), the hierarchies, and also the processes of review and acceptance of changes (Vetter, 2018). These conflicts reinforce the relevance of socio and behavioral aspects for examining the Linux kernel development model. Moreover, the high-dependence on human judgment intrinsically puts informalities, imprecisions, and unpredictability to the project development and deliverables.

This background motivates us to investigate the characteristics of Linux kernel development in light of the project community's human aspects. Bearing in mind FLOSS phenomenon transformations, we aim to capture an up-to-date picture of roles, rules, and responsibilities that support the development community. Therefore, we present in the next chapter the research methods chosen to conduct our comprehensive research in the Linux kernel development project.

Chapter 3

Research Methods and Design

According to Osterlie and Jaccheri (2007), software engineering literature has presented little about what is performed by practitioners, what activities they do, and how often they happen. More recently, Aniche et al. (2019) have claimed that "It is fundamental that the real-life experience of practitioners influences the work of researchers in the field" of software engineering. To fill the gap between practice and theory, researchers should move from a research-and-transfer model to an industry-as-laboratory approach. With this concern in mind, this work proposes a multi-methodological investigation, adopting the multivocal literature review and the ethnographic case study. For the multivocal literature review, we systematically examined grey and formal literature. Also, we performed an ethnographic case study through online participant observation in the Linux kernel project. The use of multiple methods enables to update the academia and practitioners' perception of the characteristics of the Linux kernel, the ecosystem that inspired the FLOSS development model.

In this chapter, we introduce the research methods that guide this investigation. We discuss the advantages and challenges of each of them. We also present guidelines for the use of each approach in software engineering investigations.

3.1 Research Strategies

Searching for information only in formal bibliographic references does not fit well into the universe of FLOSS. A FLOSS ecosystem usually has a community around its project, organically producing materials that describe how, when, and why a particular change occurred in the source code or the development flow. This production provides activity logs, documentation of discussion, and decision-making present in the developer's day-to-day. Scacchi and Walt (2007) define as *Software Informalism* the information resources and artifacts used by participants to describe, prescribe, or proscribe what happens in a FLOSS project. They are informal, comparatively easy-to-use, and publicly accessible narratives such as source code, artifacts, online repositories, with substantial size, diversity, and complexity not previously available globally. Online and non-academic publications can complement these materials to broaden the academic understanding of how FLOSS project developers interpret the development model of which they are part.

Besides the specificity in the production of informal narratives published openly on the Internet, FLOSS development studies must also consider the pillars that sustain developers' community for a given project. Trust and responsibility among project participants are invisible social control resources that support the open and complex development of FLOSS (Scacchi and Walt, 2007). Describing how contributors interpret the organization and workflow requires research methods capable of explaining the rationalities from an insider's point of view and puts the social, cooperative, and human aspects of software engineering practice at the center (Sharp et al., 2016). Finally, empirical studies on FLOSS are growing and expanding the scope of what we can observe, analyze and learn about these software systems and their development. Traditional research methods join new techniques for mining repositories and analysis of the socio-technical processes and environments which support FLOSS projects (Scacchi, 2010).

Considering the research questions enunciated in Section 1.3, we have a mix of question types. ***RQ1. How do software engineering studies and Linux community publications describe the current Linux development community model?*** is exploratory and descriptive. To answer subquestion *What attributes practitioners use to characterize the Linux development model? What are the current project challenges and concerns from the community perspective?*, we conduct a *grey literature review* to construct a comprehensive, up-to-date view of the Linux kernel development model considering the FLOSS community perspective. After, we analyze the selected documents using quantitative and qualitative content analysis techniques to identify the attributes used to characterize the Linux development community. For the subquestion (*Do Software Engineering studies already cover these topics?*), we chose the *multivocal literature review* with a comparative content analysis approach to assess the convergence between what is studied in academia and what is experienced in the practice of the Linux development community.

RQ1 also has an ethnographic nature because it concerns itself with the social and behavioral aspects of groups of people in the means of interactions. Community and project diversity constrain the statistical generalization of results found in case studies of FLOSS projects. However, the ethnographic approach communicates a detailed picture that enables analytical generalizations. Therefore, we include a participant-observation to explore the nuances from the daily work on the Linux kernel development process and capture the "what" and "why" of practices from different qualitative sources of data. We collect qualitative data through online participant observation in two Linux subsystems and non-participant observation of informants.

RQ2. What research techniques can be used to examine a FLOSS project through its community publications? is exploratory and descriptive. It involves deploying new methods to review non-academic literature and incrementally adapting the data collection and data analysis processes. Therefore, we describe the methods and adaptations to conduct the grey literature review on FLOSS, using the Linux kernel project as a case study. Finally, we approach ***RQ3. What are the possible gaps and opportunities for academic research in FLOSS development topics?*** by choosing specific content analysis methods to systematize concepts from FLOSS community publications and highlighting which of these concepts were already covered by studies in commercial publishers. We also compared findings from participant-observation and analyze with those found in the literature review to illustrate how to find gaps and build bridges on theory and practice

understanding of a FLOSS project development.

3.1.1 Multivocal Literature Review

A Multivocal Literature Review (MRL) is a form of systematic literature review that includes inputs from academic peer-reviewed papers and sources from the Grey Literature (Vahid Garousi et al., 2018). It aims to provide summaries of both the state-of-the-art and state-of-the-practice in a given area.

Systematic Literature Review

Systematic Literature Review (SLR) is a research methodology used to synthesize and evaluate the available evidence on a focused topic (Almeida Biolchini et al., 2007). It follows a rigorous methodology of research results to support evidence-based guidelines for practitioners (Barbara Kitchenham, Pearl Brereton, et al., 2009). The evidence knowledge allows researchers to capture gaps and concept conflicts addressed by new studies in the area (Melo, 2013).

In contrast to ad-hoc literature reviews, a systematic review follows a sequence of well-defined, strict steps following a protocol planned before execution. The whole SLR process must be documented, explicitly defining methodological steps, collection strategies, focus, research question, and intermediate results.

According to Higgins and Green (2008), a systematic review should have the following characteristics:

- A set of clearly stated goals with predetermined inclusion and exclusion criteria for studies;
- An explicit and reproducible methodology;
- The use of systematic research to reach all studies that fit the eligibility criteria;
- An evaluation of the validity of the findings of the selected studies, including the risk of bias; and
- A systematic and synthesized presentation of the characteristics and results of the selected studies.

Although SLR has become more prevalent in empirical software engineering, many of these studies use medical standards to guide reviews (Barbara Kitchenham and Brereton, 2013). Budgen and Brereton (2006) proposed a three-phase review process for Software Engineering: (1) planning the review; (2) conducting the review; (3) reporting the outcomes from the review. Almeida Biolchini et al. (2007) developed a similar three-phase template based on protocols and systematic review guidelines in the medical field to guide the planning and execution of SLR in Software Engineering. This template handles each phase of the SLR process as detailed below:

- *Review Planning:* defines question focus; question quality and amplitude, i.e., problem, question, keywords, Population-Intervention-Comparison-Outcome (PICO) components, and others semantics specificity; Source Selection (criteria definition, studies

languages, sources selection, source and references evaluation); Studies Selection (inclusion and exclusion criteria; studies types; procedures for selection).

- *Review Execution:* Selection Execution (reports primary studies selection); Information Extraction (describes extraction criteria, results, and resolution of divergences among reviewers).
- *Result Analysis:* summarizes and analyzes results using statistical methods.

According to B. Kitchenham and Charters, 2007, the main advantages of SLR are:

- Low probability of bias due to the use of a well-defined methodology (although not exempted to the prejudices of primary studies);
- Informs the effects of a phenomenon across a wide range of settings and empirical methods;
- In the case of quantitative studies, SLR enables the combination of data using meta-analysis techniques.

The SLR primary challenge is that it requires more effort than a non-systematic review. Some additional problems arise in software engineering; for example, digital libraries lack mechanisms to perform complex queries, papers omit information, and their abstracts are poor (Barbara Kitchenham and Brereton, 2013).

Grey Literature Review

According to Rothstein et al. (2005), Grey Literature (GL) is any source of information produced from academia, industry, business, and government, published in print or electronic formats, but commercial publishers do not control it. It is composed of sources of data not found in the formal literature, such as blogs, videos, white papers, and web-pages.

Alongside commercial and academic publications, GL can be used to broaden the understanding of how FLOSS developers interpret the environment of which they are part. Despite the informality of GL, it brings advantages of its own. Studies with negative or null results are easier to find in GL than in peer-reviewed academic literature, enabling a more critic perspective, with a potential reduction of bias and visualization of more balanced evidence (Paez, 2017). GL is also a leading source for identifying topics and gaps not yet covered by academic literature. It also enables investigating more up-to-date, and emerging information since academic studies incur long publication delays due to the peer-review process.

Although it is still a nascent trend, a few software engineering reviews have already included GL material (Bailey et al., 2007; França et al., 2016; Soldani et al., 2018). Nevertheless, we argue that GL material use is even more fruitful in FLOSS research, given the vast amount of public information resources produced by the FLOSS communities. In such a scenario, we claim that researchers should investigate such resources before taking the time of FLOSS contributors with surveys or interviews (V. Garousi et al., 2016), especially

considering that *i)* most of the needed information is already available with rich details and *ii)* top contributors usually do not have much time for extensive interviews.

Combining systematic review protocols with GLR helps shed light on critical industrial and practitioners concerns not yet mapped by software engineering studies (Soldani et al., 2018). However, the use of GLR also has challenges. Although systematic seview guidelines, such as Higgins and Green (2008), recommend the inclusion of this literature, a precise method for implementation is not available. Thus, GLR still faces difficulties due to the large volume of information and the lack of indexing standards and vocabulary (Godin et al., 2015).

Godin et al. (2015) present an application of systematic review methods in GLR, where information sources include GLR databases, customized Google search engines, targeted websites, and consultation with experts. They use title and source organization of documents for eligibility assessment and study selection. Finally, they search for a pre-defined set of data on each study selected.

Conducting a Multivocal Literature Review

A multivocal literature review (MRL) could improve software engineering research relevance by analyzing input from practitioner literature. Vahid Garousi et al. (2018) present guidelines to include GLR in SLR and conduct a multivocal literature review. The proposed MRL process is composed of five phases:

- **Search process:**
  - Defining search string and/or using snowballing techniques.
  - Where to search: search formally-published literature via broad-coverage abstract databases and full-text databases; search GLR via a general web search engine, specialized databases, and websites, contacting individuals directly or via social media, and/or reference lists and backlinks, and perform an informal pre-search to find different synonyms for specific topics.
  - Stopping criteria: theoretical saturation; effort bounded; evidence exhaustion;
- **Source selection:** combine inclusion and exclusion criteria for GLR with quality assessment criteria; in the source selection process of an MLR, one should ensure a coordinated integration of the source selection processes for GLR and formal literature.
- **Study quality assessment:** apply and adapt the criteria authority of the producer, methodology, objectivity, date, novelty, impact, as well as outlet control, for study quality assessment of grey literature.
- **Data extraction:** design data extraction forms; use systematic procedures and logistics, such as maintaining "traceability" links, and extracting and recording as much quantitative/qualitative data as needed to sufficiently address each research question;
- **Data synthesis:** many GLR sources are suitable for qualitative coding and synthesis; argumentation theory can be beneficial for data synthesis; quantitative analysis is

possible on GLR databases such as StackOverflow.

Crowston et al. (2012) have stated that the use of secondary data was not a research mechanism widely adopted in research on FLOSS development. Moreover, although SLR is a well-established investigative method, with steps and restrictions already defined in several fields of study, MLR, in turn, is an emerging technique, especially in the area of software engineering. Its conduction still requires adaptations for the variety of publication types on a particular topic under investigation. To perform an MLR, the reviewer should deal with the lack of robust search engines (developing customized search mechanisms), the diversity of the size, quality, and structure of the retrieved documents, and the challenge of textual and natural language analysis.

3.1.2 Ethnographic Case Study

Ethnography is a research method designed to describe and analyze the social life and culture of a specific social system (Edmonds and Kennedy, 2013). This approach's central tenet is to understand values, beliefs, or ideas shared for a group understudy from the members' point of view (Sharp et al., 2016). For this, the ethnographer needs to become a group member, observing in detail what people do and learning their language, social norms, rules, and artifacts.

In software engineering, ethnography is an appropriate method of research when one wants to understand people, culture and the social and work practices associated with them (Sharp et al., 2016). From an empathic perspective, it can explain the logic of practice from insiders' view and, thus, capture what practitioners do and why they do. Considering people's behavior an integral part of software development and maintenance (C. B. Seaman, 2008), ethnography in software engineering can strengthen investigations of social and human aspects in the software development process since the significance of these aspects of software practice is already well-established.

The ethnographic case study focuses on examining a real case in some cultural group, delimited by time, place, and environment (Edmonds and Kennedy, 2013). Its design is characterized by the intensive and holistic description and analysis of a particular social reality. It is best suited for investigations interested in exploring a group's activities rather than shared patterns of group behavior. Ethnography imposes an orientation of the investigative view on the symbols, interpretations, beliefs, and values that integrate the socio-cultural dimension of a community's dynamics (Sarmento, 2011). In this way, an ethnographic case study must present an investigative design that employs convergent methods with such orientation. Sharp et al. (2016) defines four main characteristics of ethnographic research in empirical software engineering:

- **The members' point of view:** focus on the informants' point of view, understanding what is, or is not, important, and painful for the informant.
- **The ordinary detail of life:** collect several types of data about different aspects of their informants' work and keep "open" for new possibilities.
- **The analytical stance:** provide an analysis of the results explaining how this evidence is, or is not, relevant for a particular purpose.

- **Production of "thick descriptions" for academic accountability:** provide results well rooted in meaningful aspects. Comparing and contrasting data, their aggregation, and ordering it in logical sequences to structure the knowledge.

The design of an ethnographic study must also consider five dimensions: (1) participant or non-participant observation; (2) the duration of the field study; (3) space and location; (4) the use of theoretical underpinnings; (5) the ethnographers' intent in undertaking the study.

Data Collection

Three data collection methods are primarily used in ethnographic research: participant observation, interviews, and documentation analysis (Sarmento, 2011). In this work, we collect Linux kernel project data by examining the community-produced documents and the participant-observation in two Linux subsystem communities.

Participant and Non-Participant Observation

Data from direct observation contrasts information obtained through interviews and questionnaires. The reason for the differences is that humans do not always do (or will do) what they say they have done, especially when it involves their reputation. In this case, the participant observation method makes it possible to explain the meaning of the observed fact's experiences through the observer of the fact experienced (Robson and McCartan, 2016) and allow the informant to judge what is essential rather than what she thinks is essential.

An ethnographer spends her time working, discussing, participating, and living, interacting with informants. She becomes a member of the experimental group, interpreting what is happening around her. Hence, in addition to sensitivity, the observer needs the personal skills necessary to conduct participant observation. In a FLOSS community, the software engineer has the facility to assume participant-observer's role and provide valuable information and practical consequences for software practice. Also, informants of the same culture and living in similar conditions generally reduce the time required to complete an ethnographic study (Sharp et al., 2016).

According to C. B. Seaman (2008), a considerable part of software development work is not documented because it occurs within the developer's mind. Software developers find it easier to reveal the processes present in their thoughts when communicating with other software developers, making this communication a valuable opportunity to observe the development process. In this context, a data collection method is the continuous observation of a developer, recording all interactions with the other group members through notes or journals. This journal should be kept confidential throughout the study to preserve the observer's total freedom of writing. Each note should include the location, time, and participants of the observation, the discussions that took place, the events during the observation, and the tone and feelings involved in the interactions. As the observer becomes more familiar with the group studied, the notes gain a wealth of detail.

3.1.3 Data Analysis and Crossing Information

Qualitative data are those in which information is expressed through figures or words, whereas categories or numbers represent quantitative data (C. Seaman, 1999). An approach to extracting quantitative variable's values from qualitative data is the process of *coding*. Coding makes it possible to get more precise and reliable quantitative data and quantify subjective information to perform some quantitative or statistical analysis. To avoid loss of information in this transformation, the researcher needs to be aware of the variations of terminology in describing the phenomena, the different ratings for the same subject, and the margin of reliability between them.

Three processes describe the coding analysis (Edmonds and Kennedy, 2013):

1. *Open coding* identifies categories of information about the phenomenon being examined.
2. *Axial coding* involves taking each one of the categories from the open coding and exploring it as a core phenomenon.
3. *Selective coding* systematically relates the core category to other categories.

In this work, we consider distinct approaches for qualitative content analysis: conventional, directed, and summative (Hsieh and Shannon, 2005). They differ by how the initial codes are developed. When an initial coding scheme exists, it is revised and refined by interpreting the contextual meaning of specific terms or developing additional codes.

1. *Conventional content analysis* derives code categories during content analysis and directly from the text data.
2. *Directed content analysis* starts from initial codes defined by a previous theory or relevant research findings and improves them during data analysis.
3. *Summative content analysis* starts by counting and comparing keywords derived from interest of researchers or review of literature. Additional analysis proceeds to interprete the underlying context.

These different approaches also support crossing information. Crossing and analyzing information obtained from different sources, different types of data, or different collection methods allows performing a Triangulation (C. Seaman, 1999). It is considered the most powerful method of "confirming" the validity of conclusions. In the case study, triangulation is essential to prevent the unilaterality in a piece of observation, a testimonial, or a document that could compromise the understanding of reality (Sarmento, 2011). Different kinds of measurements provide repeated verification; however, triangulation gives reliability rather than validity information. This method also reveals inconsistency or even direct conflicts, helping to elaborate our findings or initiate a new line of thinking (Miles, 2015).

The crossing of information obtained through the three data collection methods mentioned above (GLR, SLR and Participant-Observation) can provide a comprehensive and current view of the characteristics of the Linux development model. Following our workflow (Figure 1.1) and the methods described here, the next two chapters describe the process of data collection and analysis according to the research phase and the information

source. We present our findings from each investigative stage and how we combined them to reach the study goals. Due to the lack of definition on grey literature review methods, we also provide a set of strategies to examine FLOSS community publications. Finally, we discuss our findings considering three perspectives: academic studies, community publications, and development community participation.

Chapter 4

The Linux Kernel Development Model from the FLOSS community perspective

The inclusion of GL in systematic literature reviews has been increasingly widespread in the medical field (Godin et al., 2015; Paez, 2017). However, in software engineering, this procedure is not easily found yet. For that reason, we relied on guidelines from studies in both areas (Soldani et al., 2018; Vahid Garousi et al., 2018; Godin et al., 2015; Paez, 2017) to structure a search plan suitable for FLOSS publications.

To address the lack of well-established methods specifically tailored for exploring grey literature, we adapted the existing systematic review methods to examine community productions with defined steps and constraints. Systematic Literature Review (SLR) is a research methodology that follows rigorous procedures to synthesize and evaluate the available evidence on a focused topic (Almeida Biolchini et al., 2007), supporting the development of evidence-based guidelines for practitioners (Barbara Kitchenham, Pearl Brereton, et al., 2009). Combining SLR methods with GLR helps shed light on important industrial practices still not mapped by conventional software engineering studies (Soldani et al., 2018).

As with SLRs, the GLR methodological plan provides guidelines, structure, and transparency to the search methods (Godin et al., 2015). However, due to grey materials' characteristics, GLR searches are often less methodical than traditional systematic searches for academic publications. The researcher should map data sources and types, search terms, selection criteria, and boundaries to reduce searching bias. This approach also helps manage the variety of search terms and the volume and diversity of "grey" materials available throughout the web.

Accordingly, this chapter presents the design and application of the systematic grey literature review on the Linux kernel development model. We highlight the challenges, specificities, adaptations, and lessons learned from our experience examining publications from the FLOSS community. This chapter is an extension of our previous work on defining strategies to examine GL documents of FLOSS projects (Wen, Leite, et al., 2020).

Section 4.1 describes a set of strategies and guidelines to support grey literature reviews on FLOSS. Section 4.2 presents a mind map resulting from our content review and analysis strategies to systematize the concepts that describe the current Linux kernel community model. Section 4.3 explains how the selected documents from the grey literature provide the concepts outlined in the mind map.

4.1 Systematic Grey Literature Review

We present here guidelines for supporting future FLOSS literature reviews. We mapped challenges and adaptations for GLR on FLOSS and summarized our GLR flow in Figure 4.1. It is a first effort in defining a methodology for conducting GLRs for software engineering research.

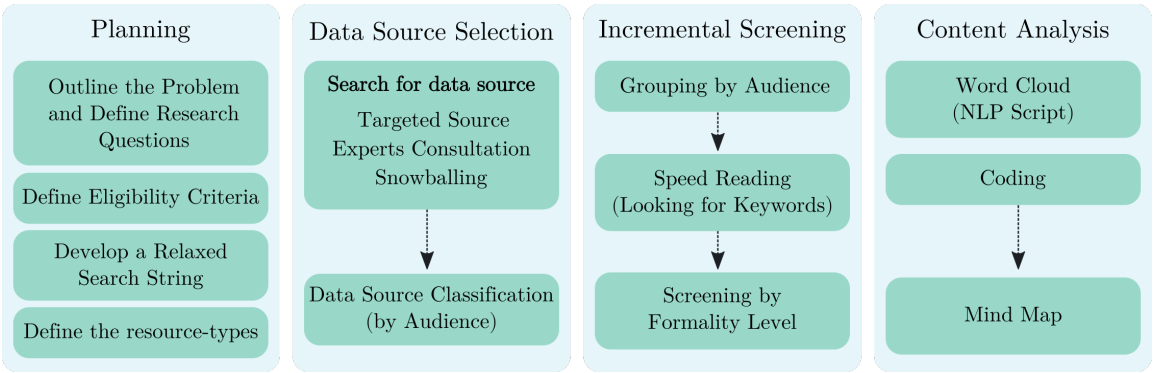


Figure 4.1: *Steps of our GLR Flow*

4.1.1 GLR Planning

The GLR Planning covers four essential steps to ensure the quality of the selected documents:

1. Outline the problem and define the research question.
2. Define the inclusion and exclusion criteria.
3. Develop a relaxed search string.
4. Define the resource-types to consider.

Also, we developed and applied additional methods to refine the screening step and complete the GL search. From an extensive selection of information sources, we evaluated and selected most relevant data sources. After, we identified and examined documents using an incremental screening process. In the following, we describe better the steps of planning GLR on FLOSS, and conduction of the screening process.

Outline the Problem and Define the Research Question. The first step in a typical GLR planning process is to establish the need for this type of review. Researchers should then describe the study's background and objectives in a protocol (Higgins and Thomas, 2019) to delimit the scope of research and define research questions.

The distributed, community-based development of the Linux kernel project provides a wealth of alternative resources of professionals' information. This materials examination can ensure the most current picture of the Linux kernel development model and expand the knowledge on the FLOSS state of practice. For example, some organizations periodically provide reports about the State of Linux Kernel Development, which promptly discusses the Linux development statistics, who is working on the kernel, sponsoring the development, and so on. In this regard, the research question *"RQ1. How do software engineering studies and Linux community publications describe the current Linux development community model?"* guided our review case. At this phase, we first examined the perspective of community publications.

Define Eligibility Criteria. Reviewing the GL poses many validity risks since these materials are not reviewed or provided by traditional publishers and channels. Besides this, evidence in them is often based on personal experience and opinion, at the risk of catching fake news and views from people outside the problem set. Based on the experiences reported by Soldani et al. (2018) and Vahid Garousi et al. (2018), we recommend defining inclusion and exclusion criteria and combining them with additional control factors to mitigate this problem.

Table 4.1: *Grey Literature Review - Eligibility Criteria and Control Factors*

Eligibility Criteria	Control Factors
Published between 2009 and 2019.	The document is publicly accessible.
Most current version of the document.	The content is not centered on: technical issues or specific features.
Available in English.	
Published online by institutions, industry-oriented magazines, and practitioners of the FLOSS area.	Historical facts, statistics, or Linux kernel project artifacts support the statements in the document.
Describes developments in the Linux kernel by: (1) reporting practices; (2) presenting statistics; (3) discussing management and rules inside the project; (4) or studying the project development and/or its community.	The document is published by: (1) a reputable organization or magazine; (2) an individual author associated with such organizations and magazines, or (3) a practitioner with more than five years of FLOSS experience.

For our review, we selected documents that satisfied the eligibility criteria and also some control factors to assure sample quality, as presented in Table 4.1.

Develop a Relaxed Search String. Defining a flexible set of search terms, based on the problem definition and eligibility criteria, is essential to deal with the variety of terminologies used by persons and institutions for a given study object, since some popular keywords in software engineering studies are still not commonly used by practitioners. Besides, a GL writer has little concern with standardization and structuring. Consequently, they usually use more informal and heterogeneous terms. Another issue is related to

applying the search string in a website search engine. Much of these tools cannot address both diversity and lack of text structure. Therefore, even with synonyms' prediction, we observed that broader and simple strings were essential to reach potentially relevant documents during the search process. Finally, the diversity of the audience of each data source required a warm-up round to understand how an organization or person commonly refers to search string elements and evaluate the coverage of these terms accordingly.

For the Linux project case, we first defined a base search string, *Linux development model*. Then, we broke it into words and listed potential synonyms for each word, as follows:

1. **Linux** → Linux Kernel;
2. **Development** → Business, (empty string);
3. **Model** → Scheme, Rules, Process, Guide, Community, Culture, (empty string).

Define the Resource-types. Considering the great diversity of GL resources, list the types of data to be considered for analysis. Then, separate this data into *in-text content* and *audiovisual content*, since the latter usually demands more effort to inspect than the former. Based on the categorization in "shades of GL" presented by R. J. Adams et al. (2017), rank the resource types according to expected formality levels. This rank should guide the document review order and support the development of appropriate stopping criteria for the screening phase.

In our case, we discarded audiovisual resources due to the effort required to examine this content and the transcription process's difficulty. We also assessed the expected quality of textual contents from their formality levels. We decided not to include mailing-list e-mails from this assessment because these records often have opinion-based discussions. Moreover, in the Linux kernel project, mailing-list e-mails play dual roles: text and code (artifacts) community production. Nevertheless, mailing-list e-mails can be an interesting venue for future research.

4.1.2 Data Collection

Even in systematic reviews, changes in a review protocol are sometimes necessary to adjust methods to unanticipated circumstances (Higgins and Thomas, 2019). Despite all efforts to adhere to the search plan, we used some strategies to overcome unexpected limitations and difficulties during document screening.

Data Source Selection

In a review of the traditional literature, data source selection often consists of a set of similar digital libraries from prestigious organizations in academic computer science publications or similar. However, for GL of FLOSS projects, data sources are not so obvious, and selecting them requires specific knowledge and consultation of experts.

A good data source selection minimizes GLR drawbacks such as time spent and omitting significant evidence. To select GL suitable data sources for FLOSS studies, the researcher

should carefully consider the initial search string and the content types. We also recommend four strategies for extensive searching data sources:

1. **Search for targeted source:** Conduct a web search to identify organization webpages, magazines, and relevant blogs on FLOSS development. Use the advanced search tool to restrict results to English documents only. In this step, we evaluated the first ten pages of search hits and manually inspected each website for data source selection. We also evaluated blog posts that list relevant pages for those interested in tracking Linux project updates.

2. **Consult experts:** Consultation with experts can help discover other relevant data sources. We consulted four of them and identified additional data sources not previously covered.

3. **Apply snowballing:** Add data sources if links or references to their contents repeatedly appear during the sample selection phase.

4. **Classify the data sources:** The data sources should be classified according to their proximity to the object of study, in our case, the Linux kernel development. We divided data sources into sources that deal specifically with the Linux kernel and references that deal more broadly with IT or FLOSS subjects. The publication collection did not follow any order of priority; however, to improve the performance of the review process, a researcher should first evaluate contents in the specific data sources, and use data from the other sources as complementary information.

Since search engines use unique algorithms to generate their relevance ranking schemes, the researcher must be prepared to address each data source's singularities. Accordingly, we recommend a prior evaluation of data sources in terms of audience, text structure, search tools, quality of search results display, and updates. This step helps to identify the main subject addressed, potential search pitfalls, and the formality of documents.

For our review, the mentioned data source search strategies reached 15 data sources: eight magazines, three developer blogs, two practitioners-oriented news websites, one institutional website, and one wiki page. We evaluated each of them and discarded one for not allowing open access to publications, as presented in Table 4.2. We flagged two others to receive proper procedures in the evidence collection and analysis phase. One had a restricted time interval (it was deactivated in 2015), and the other had its content only partially screened, as not everything was openly available.

Some data sources did not provide a search tool, and others had a varying quality of features. The main issues faced were:

1. site pagination only worked until page 10;

2. the date of the indexed article in the search was not the same as the date of publication;

Table 4.2: *Database evaluation and selection*

Name	Type	Issues	Selected
wire.com	Magazine	Poor search tool Browsing results break after page 10	No
drdoobs.com	Magazine	Inactive since 2015	No
linuxformat.com	Magazine	Need payment	No
linux-magazine.com	Magazine	Some contents need payment	No
linuxjournal.com	Magazine	The timestamp displayed in the results is not the same of the article publication	No
linux.com	Tech website	Cannot sort by date; no abstract Many hits are only external-publication links	Yes
linuxfoundation.org	Org website	Layout in-grid/block, not in-list No pagination; No option of sort by No filter; No abstract Many hits are only external-publication links Some external links are broken Need a google search to find the document	Yes
cnet.com	Magazine	(not evaluated)	No
networkworld.com	Magazine	No regex filter, results is "or" inclusive Too many results; required to sort by relevance Search until page 20	No
opensourceforu.com	Magazine	Search string is "and" by terms Less accuracy of search strings	No
kernelnewbies.org	Wiki page	Good search tools and enable regex “Linux kernel” more accurated than “Linux”	Yes
lwn.net	News site	Search for "and" of terms (no order) or "or" Sort by date; Return only the last 500 publications Articles are written by experts	Yes
kroah.com	Blog	No search tool Manual Inspection	Yes
blog.ffwll.ch	Blog	No search tools, but have tags Manual Inspection + Tags	Yes
sage.thesharps.us	Blog	Manual inspection + searching for the word "Linux"	Yes

3. it was not possible to filter or sort by date;
4. results were shown in blocks rather than lists;
5. there was no pagination;
6. did not present any results ordering;
7. did not allow filter searching;
8. did not enable search by an expression, only search by any of the words of the string;
9. search for all words in the string, but in arbitrary order;
10. limit on the number of documents returned.

To overcome weak search mechanisms, we recommend the following strategies:

1. **Choose a suitable order of search hit:** by date, when it is possible to search by

an exact sequence of words; by relevance when the tool only seeks for any of the terms in the string.

2. **Define stopping criteria** influenced by weak search engines or large volumes of data, such as page number restriction, a suitable number of hits, the decline in quality, and availability of evidence.

3. **Consider searching by tags/categories**, whenever possible.

A researcher should perform a manual inspection of publication titles and dates when a data source does not provide any search engines but had great data potential (such as some blogs of experienced developers or important personages).

Grouping and Screening Phase

We have developed an incremental method for handling the number of search hits and the lack of standard indexing. This method consists of dividing the screening process into data source audiences and iteratively conducting the sample selection. In each iteration, the inspection becomes more detailed and follows the ascending order of document formality.

For our review of Linux kernel publications, we separate the selection process into two, according to data source audience: the most specific Linux source documents first and then the most widespread for the IT industry. We documented this screening process in a spreadsheet[1] divided into two tabs: (I) Linux-oriented publishers (8 data sources); and (II) IT-Industry-oriented publishers (7 data sources). In the first tab, we identified 236 documents of publishers and practitioners experts on Linux. In the second tab, we collected 104 documents published in magazines and blogs of the general IT industry subjects. Each spreadsheet row represents a potentially relevant document. Seven columns index the document data: (1) date of last update or access; (2) title; (3) authoring organization or person; (4) category of the data source; (5) URL; (6) search strategy that located it; (7) links or references in the document.

Incremental Screening Process

After classifying the data sources by their target audience and separating the documents into two spreadsheets, we started the identification and incremental screening phase. Two reviewers carried out this phase to mitigate bias issues. Reviewer 1 is the leading researcher of this work and Reviewer 2 is external to the investigation but familiar with the object of study. Both have recent development experiences in the project selected for study.

GL rarely presents standard indexing and controlled vocabulary, with a good quality title, abstract, or keywords to assist in the document's inclusion or exclusion. Hence, for this phase, we opted for an incremental exclusion process. First, remove duplications or documents with only a link for another, bringing the sample's linked material. Second, try to evaluate titles, looking for any relation to the object under study.

[1]Data available on gitlab.com/ccsl-usp/glr/-/raw/master/linux-case-data.pdf

However, as GL titles are often not accurate or "attractive" but misleading, this process was not very productive. Because of this, we opted to **speed read the full-text**, searching for the keyword *Linux* to quickly understand how the project was addressed in that document and evaluate the exclusion criteria. When the eligibility criteria were not clear, we kept the document on the list. Finally, we classified the material according to the categorization of GL of R. J. Adams et al. (2017) and performed the full-text screening following this order of priority: (1) less informal literature first; (2) maintainers blog posts; (3) reports of project landmarks and emblematic cases; (4) documents from the snowballing process.

Table 4.3: *Number of documents selected per database*

Database	# Initial Selection	# Selected documents
Linux Foundation Website	77	25
Linux.com	69	40
LWN.net	31	21
Digital Magazines	9	3
Kernel Newbies	21	7
Members Blog	29	12
Total	**236**	**108**

As a result, Table 4.3 shows the distribution per database type of the 108 selected documents that covered topics related to the Linux kernel community model.

4.1.3 Content Analysis

The screening process resulted in the selection of 108 documents. The lack of standards in text structure and terms used is a visible problem in documents of this type. Thus, we resort to some strategies to deal with the challenge of a sample diverse in size, structure, and formality:

1. We looked at the data from a word cloud structure.

2. Using the same formality classification from the screening process, we coded the most formal documents (in this case, the foundation's annual reports) in order of antiquity. We plotted the result in a mind map. With concepts separated in areas/categories, we verified a poorly characterized area that needed more depth.

3. Following the formality level strategy, we grouped the documents and ordered them by recentness to complete the open-coding process.

Below, we detail these strategies and the techniques used in conducting the grey content analysis.

The Word Cloud

To explore the usage of words and content in community publications, we started the content analysis using a summative technique to identify and quantify the appearance of words in the textual data from selected documents (Hsieh and Shannon, 2005). We developed a script to build a word cloud from all selected documents' text content, searching for familiarity and automatization. Building a word cloud was just a warm-up to be familiarized with standard terms, Linux community concepts, and aware of the volume of information. The script[2] was developed in Python 3. The script uses a library of natural language processing named Spacy[3] to tokenize the nouns and proper nouns. Spacy is open-source software and good for beginners. It provides pre-trained models for several languages, but, for our sample, we used a small English model trained on written CNN web text (blogs, news, comments).

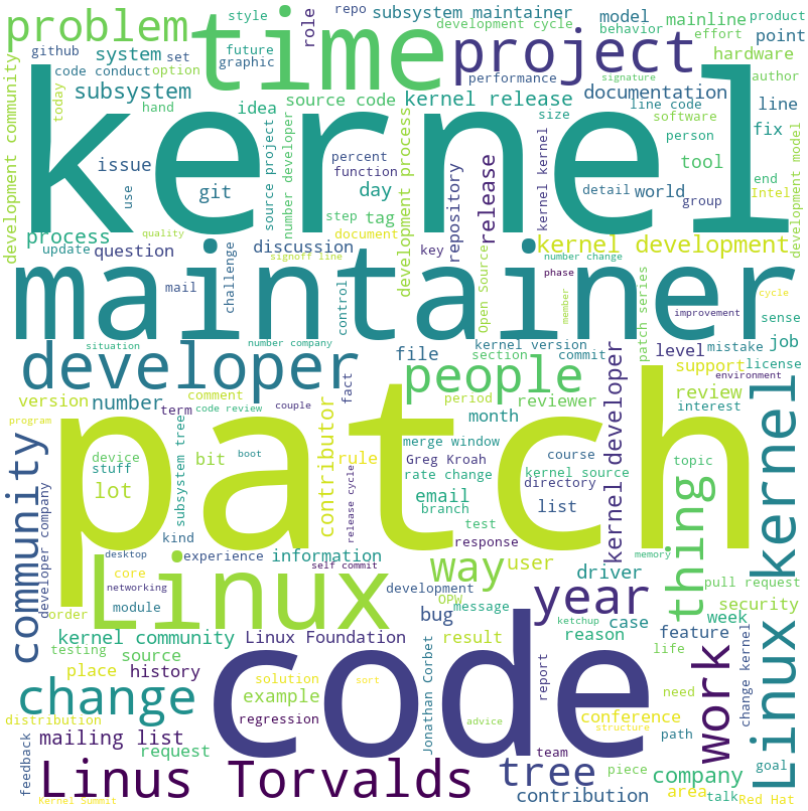


Figure 4.2: *WordCloud from Grey Literature textual content*

We used a word cloud generator[4] to count word frequency, remove stop words and plot the result as a word cloud. With this visual representation of the text data, we could identify relevant words and potential core concepts to facilitate the open-coding. Figure 4.2 present the automated WordCloud from the raw text of the 108 selected documents. We can identify words related to the community's elements, such as the nouns maintainer, developer, people, contributor, company, behavior, discussion, code of conduct, reviewer,

[2]Script available on https://gitlab.com/ccsl-usp/glr/-/blob/master/script_wordcloud.py
[3]https://spacy.io/
[4]https://github.com/amueller/word_cloud

conference; and prominent proper nouns Linus Torvalds, Greg Kroah, Jonathan Cobert, Linux Foundation.

We also grouped words related to development artifacts: code, patch (unit of code changes on a source-code version), source project, device, repository, documentation, kernel version, source code, lines of code. The word cloud also reveals process-related codes, such as merge window, pull request. Finally, some codes characterize the development and the community structures: subsystem, subsystem tree, subsystem maintainer, mailing list, email, mainline, role.

Counting the frequency of specific words and visualizing potential core labels in a word cloud is the initial part of our coding approach. Using a computer-automated method to reveal the most common and potential concepts helps avoid bias and misleading information. However, understanding the contextual use of each uncovered concept also includes the process of content interpretation. Therefore, after identified patterns in the data, we obtained potential labels that become the initial coding scheme. The remaining content analysis used a more conventional approach to discover the range of meanings that each code presents in the data from this initial coding scheme.

Open-coding

Since we are developing strategies to analyze grey literature materials in a distributed group, we use some online tools to support good documentation of our open-coding process.

We used online spreadsheets from the Google Docs Editors office suite and kept the data collected in a single sheet, adding a new column to each new step of both screening and analysis. We resorted to the feature of "Suggests" to document the coding process, i.e., which words were discarded and how a label was extracted from a sentence, without excluding any word of the text fragment. Finally, we developed three versions of the mind map. First, to become familiar with the data, second, to set up a basis to relate concepts, and finally, we extended the map to cover all the coding concepts. We used the Coggle[5] platform for mind maps and we chose on Color Brew[6], a colorblind safe color scheme for qualitative data. All figures and mind maps in this Chapter 4 and in Chapter 5 subsequent follow the same selected color scheme.

Table 4.4: *Grey scale of publications*

Grey Scale	Types of Publications	#
1	Project Development Reports	4
2	Foundation website articles and white papers	21
3	Magazine and news website articles	64
4	Community Wikipages	7
5	Blog posts	12

We organized and grouped codes into meaningful subjects and produced a conceptual map, considering the initial coding scheme and codes obtained by a random selection

[5] https://coggle.it
[6] https://colorbrewer2.org

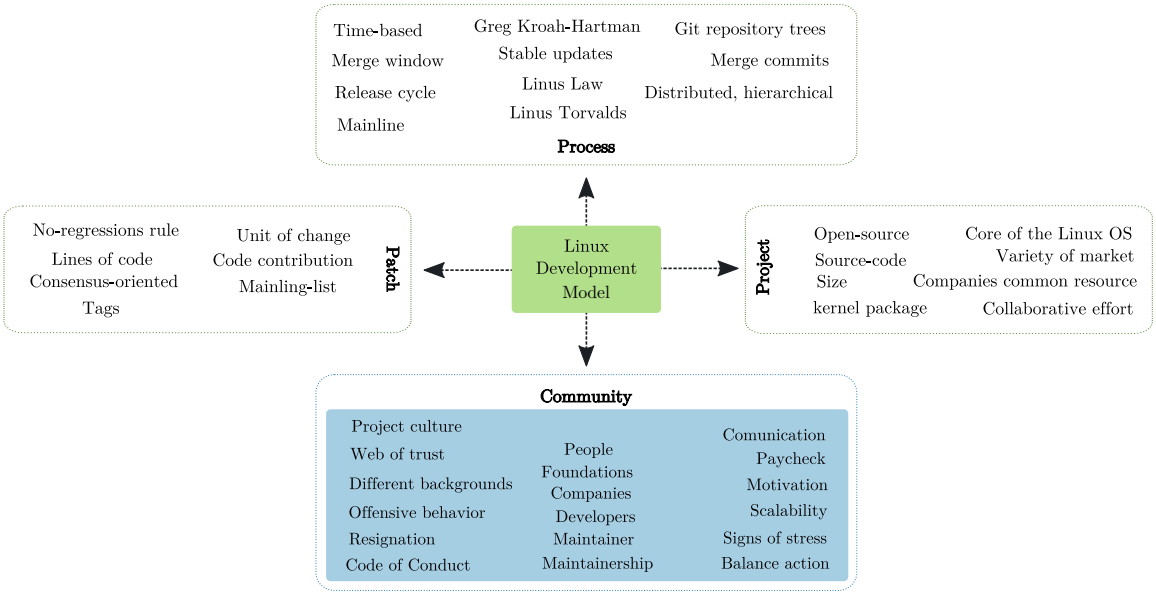


Figure 4.3: *Grouping concepts from selected documents by subject area*

of materials from different formality levels. This conceptual map revealed what topics are under discussion in the selected community publications. Figure 4.3 shows four key areas – Process, Community, Project, Patch – and a sample of codes related to them. We observed that many areas present a solid understanding with enough saturation. The Linux kernel development process is well-documented and systematized in the project Documentation (Documentation, 2019) and other publications with this propose (Corbet, 2008). however, concepts concerned with community aspects are scattered around the documents, lacking systematization. Therefore, we decided to focus on the Linux kernel community's concepts to build a comprehensive mind map.

We classified the selected documents for levels of formality (greyscale), as shown by Table 4.4. We started to open-code content from the four most formal documents - Linux Kernel Development Reports published by Linux Foundation, ordered by antiquity. We sorted codes into categories and built a base map version to organize them into a hierarchical structure. From this, we coded the remaining selected documents and mapped the concepts in batch. Finally, we define each category, subcategory, and code. We report our findings in Section 4.2. We also exemplify how code and category are identified from the data by quoting text fragments.

4.2 The Linux Kernel Development Community Model

The strategies just presented in this chapter to conduct GLR on FLOSS answer *RQ2. What research techniques can be used to examine a FLOSS project through its community publications?* We analyzed content from selected documents to identify the characteristics used in community publications to describe the Linux kernel development community. In this section, we present the results obtained that answer *RQ1.2. What attributes practitioners*

use to characterize the Linux development community? What are the current social and organizational challenges from the community perspective?

Even focusing on one of the meaningful subjects on the Linux kernel community, we reached an extensive mapping of the community characteristics and critical elements. Therefore, we narrowed the codes obtained in a mind map of categories to highlight relevant topics from the community publication perspective that should be addressed by future FLOSS research.

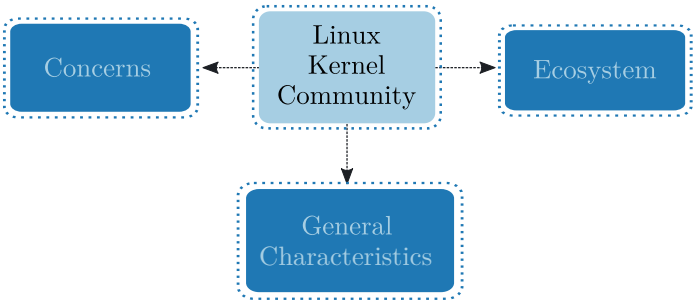


Figure 4.4: *The three fields of the Linux Kernel Community mindmap*

As presented in Figure 4.4, we sorted the concepts from the grey literature in a mind map comprising three fields: General Characteristics, Community Ecosystem, and Community Concerns. This mind map resulted from the exploratory examining of 108 documents published by practitioners and organizations inserted in the Linux kernel community. Although we have a vast number of documents, many are shorter and less-structured than high-level academic papers and required some expertise and side search for understanding and extracting the correct knowledge.

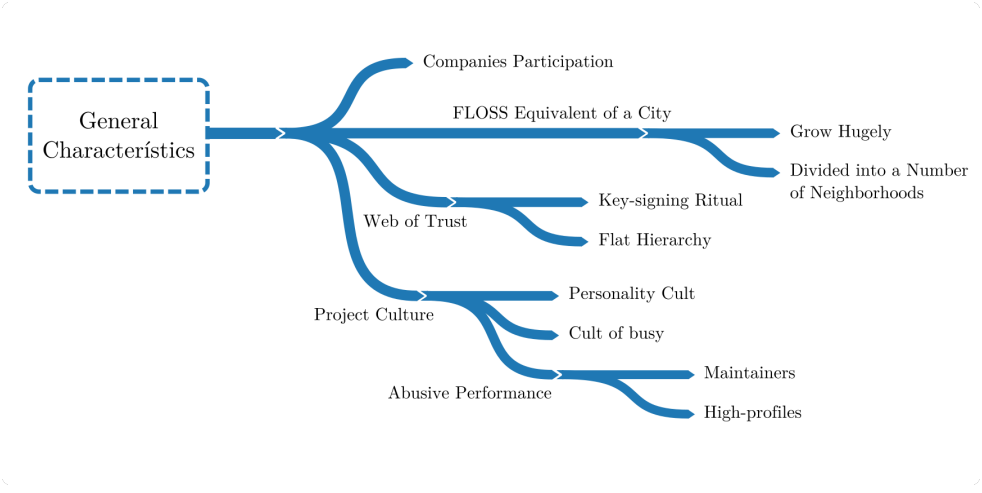


Figure 4.5: *Mindmap – Attributes about General Characteristics of the Community*

For readability reasons, the entire mind map is available online in a git respository[7]. Here, we have divided it into four figures corresponding to the three fields just discussed,

[7]Mindmap available on https://gitlab.com/ccsl-usp/glr/-/blob/master/Linux_KernelDevelopment_Comunity_Categories.pdf

where the Community Ecosystem was subdivided. General Characteristics are presented in Figure 4.5, dealing with the community as a whole. The Ecosystem is presented in two parts: Figure 4.6 (companies, foundations, and people) and Figure 4.7 (developers, their roles, and organization). Finally, Concerns, as seen in the selected community publications, are shown in Figure 4.8.

Figure 4.5 describes general characteristics and summarize transversal features of the Linux kernel community. The community creates a common project culture. Its participants are organically organized in a web of trust, supported by features on the git version control tool. The project development is divided into subsystems or slices analogous to city neighborhoods. Finally, the project diversity of usage attracts companies and people from various markets, countries, and cultural backgrounds.

The remaining figures are shown on the following pages. Figure 4.6 presents three key entities that compose the Linux kernel development community: companies, people and foundations. Each of them has a specific interest, and their participation in the community may influence the development process. Also, the map describes how people in the community structured themselves according to roles and responsibilities, and what are their standard means of interaction. Finally, in the sequence of attributes on the profile of who directly participates in coding changes, the developers, Figure 4.7 approach the members' path into the community, their roles in the work structure, their identity and affiliation, and their motivation (reasons to stay in the community).

The community publications also expose practitioners' concerns about the weaknesses of the current project process and management. They commonly express opinions, questioning the state-of-the-practice and bringing potential solutions for everyday issues and conflicts. Figure 4.8 depicts what has bothered the community for the past ten years. They report issues ranging from conduct norms on communication to scalability weakness on the development process. These problems may affect the Linux kernel development since developers' interaction supports project sustainability. Consequently, any approach to mitigate or overcome difficulties on the collaborative work has the potential to change the Linux development model, such as maintainership, hierarchy, grants, and process of review.

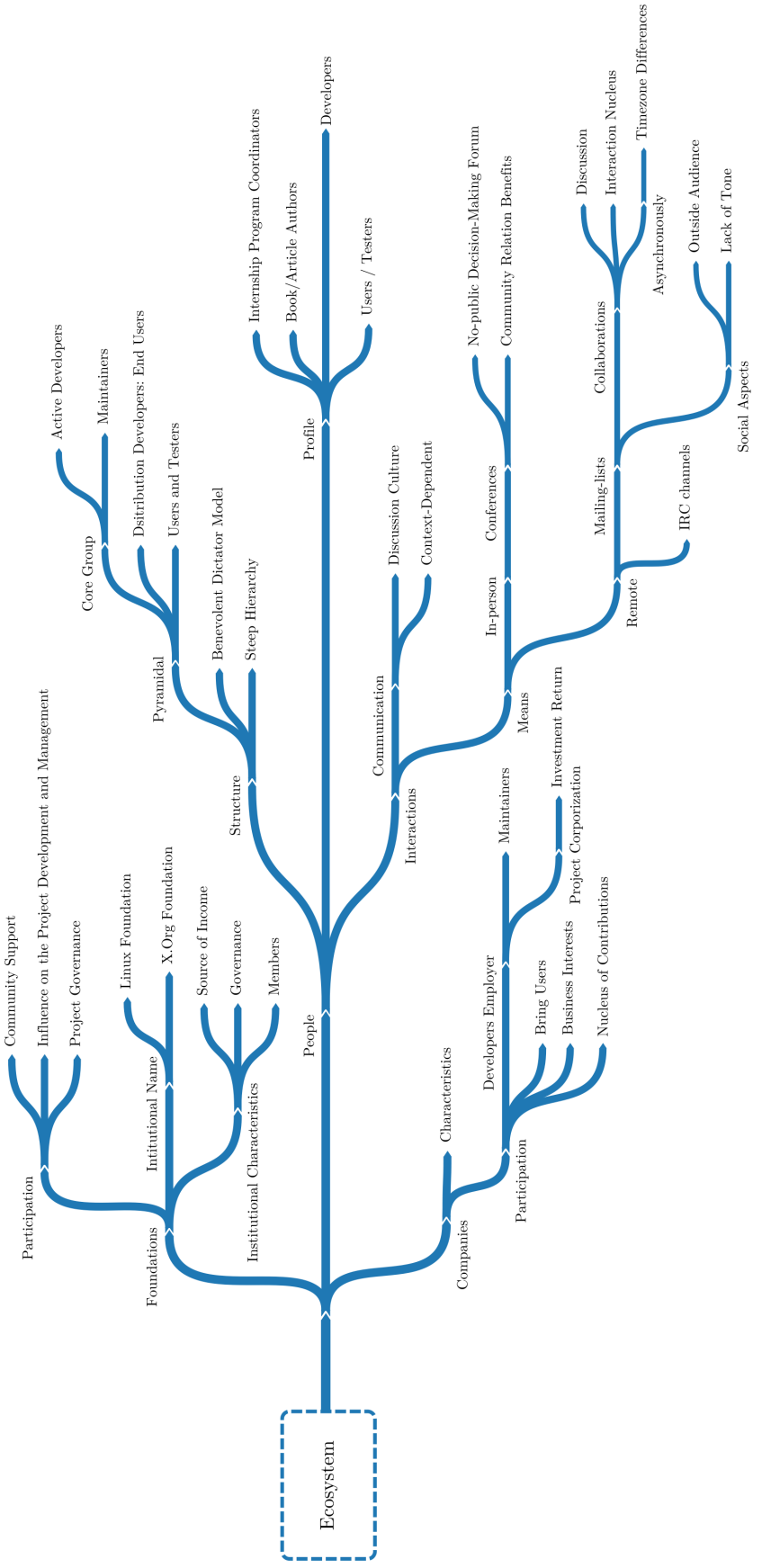


Figure 4.6: *Mindmap – Attributes about the Community Ecosystem (1)*

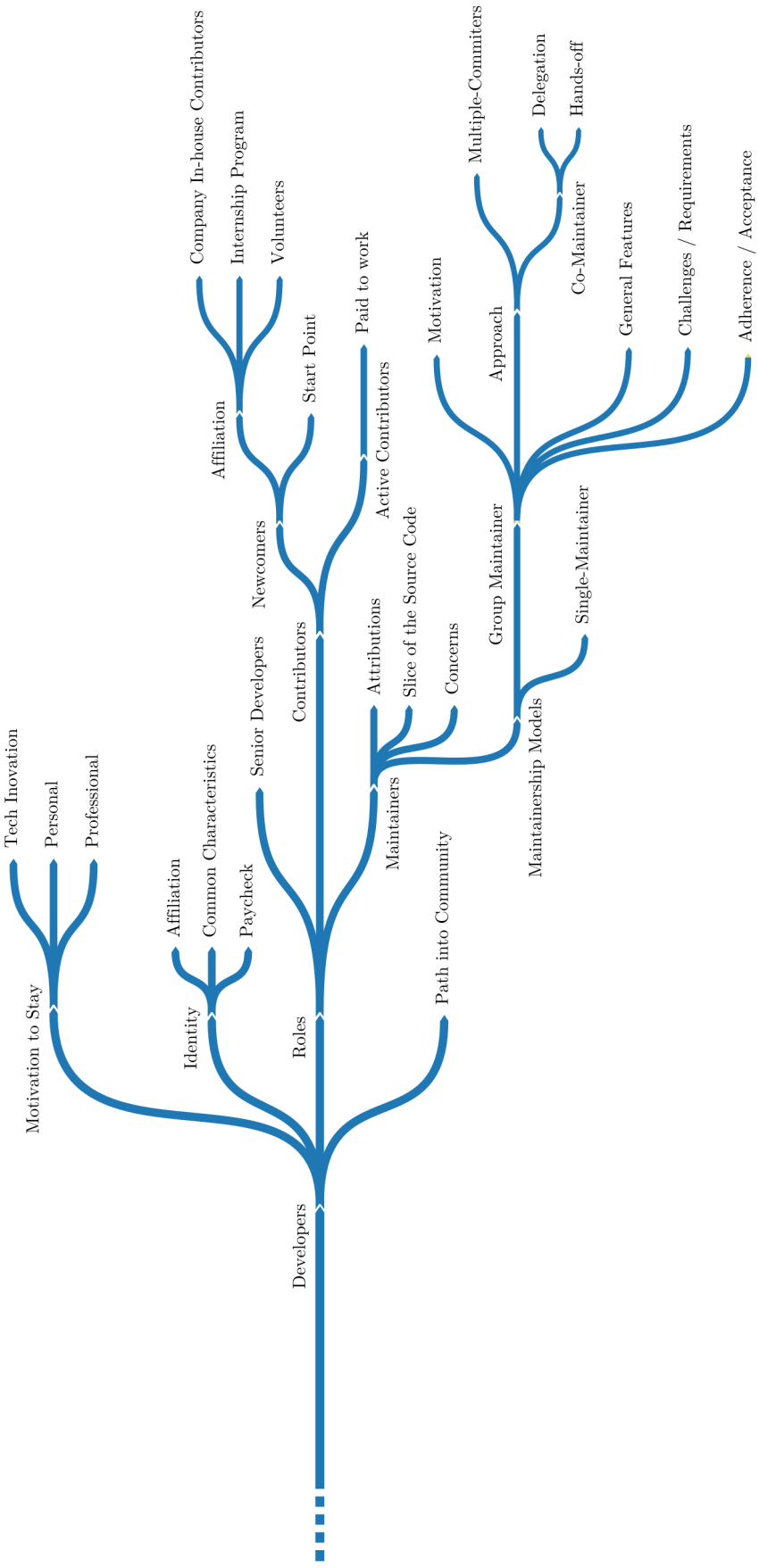


Figure 4.7: *Mindmap – Attributes about the Community Ecosystem (2) – Developers*

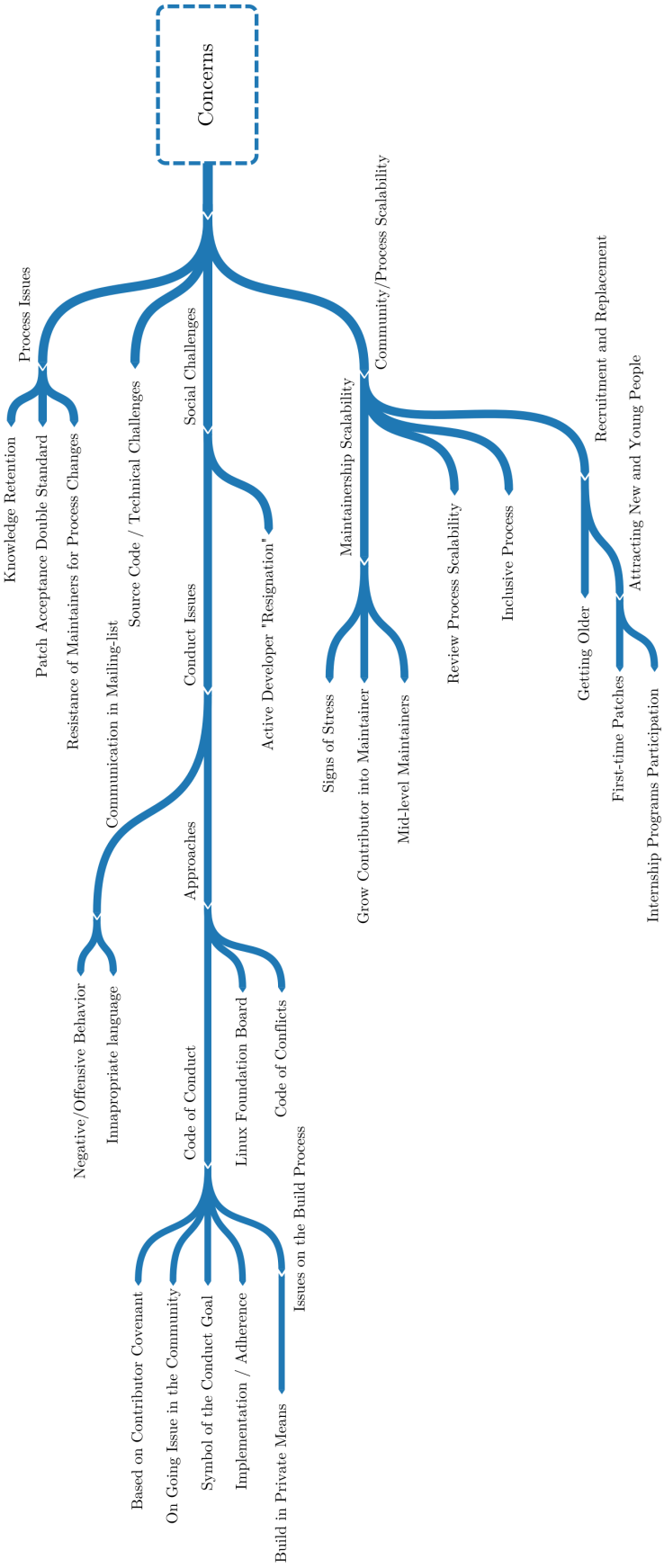


Figure 4.8: *Mindmap – Attributes about Community Concerns*

4.3 Linux Kernel Community Attributes Described by Community Publications

This section explains how the community claims presented in the 108 documents analyzed[8] is connected to each concept presented in the mind maps of the previous Section 4.2. We strongly recommend that the reader use the figures referenced in each section to guide the reading of each concept's description. For a better understanding of what comes next, the concepts were formatted in the text as follows. Categories are displayed in two levels: as subsections in bold or underline. We include some text fragments from the selected documents, in italic and within quotes, to enrich the discussion on the attributes, some of which are highlighted in bold.

4.3.1 General Characteristics

In the past two decades, the FLOSS ecosystem has diversified its forms of development. Also, the development of the Linux kernel has undergone notable transformations. From the governance perspective, the Linux kernel development shows aspects that diverge from those that inspired an anarchical, bazaar-like model and converge to dictatorship or a kind of federalism.

Much of the changes in the development of the kernel are a consequence of the growing **companies participation**, many of them large corporations. *"Corporate participation in free-software development isn't going away, or, at least, so we must hope. But we have to try not to sell out to it entirely. A crucial piece of that is not allowing any single company to control any important project on its own"* (Corbet, 2016a).

The source code and the number of system resources are also growing. As a consequence, a scalable organization of development work is crucial for the sustainability of the project. A community of developers does this work, where the Linux community is seen as the **FLOSS equivalent of a city**. It **grows hugely**, and the source code is **divided into a number of neighborhoods** with different levels of friendship between them. As each neighborhood has its own local rules, many bad experiences happen because someone crosses over to a new neighborhood and violates one of the rules.

The pieces of the community interact with each other through a **web-of-trust**. The web-of-trust follows a **flat hierachy** with a core group of subsystem maintainers and few mid-level maintainers. The patch flow into the mainline is a good representation of the maintainers level structure, *"a structure that is far flatter than the hierarchical maintainer model would suggest. In the real world, mid-level maintainers are relatively rare; most maintainers send pull requests directly to Linus Torvalds."* (Corbet, 2017a). Changes from each subsystem repository is merged into the mainline following a **key-signing ritual** to contribution verification. Subsystem maintainers, responsible for a piece of the source code, request to merge changes in their repository by signing of pull-requests. This is supported by a git feature of GPG-sign commits and tags. This approach aims to

[8] All statements made in this section are summaries of the content present in the selected documents. References are available at https://gitlab.com/ccsl-usp/glr/-/blob/master/GLR_Linux_kernel_development_model.ods

improve security and authenticity when merging repositories, but the community says it is inefficient, as it is not mandatory and maintainers rarely ask for signatures.

Over time, the community creates a **project culture** related to commonly characteristics, expectation and values of members. The intense work of maintaining the good performance and supporting new features insert the **cult of busy** in the community. There is also a certain **personality cult**, where high-profiles and maintainers could cross the line with abusive performance. This is solidified because, despite the changes over the thirty years, the top-level is composed of the same long-standing members with few renewals.

4.3.2 The Community Ecosystem

As mentioned in Section 4.2, Figure 4.6 and Figure 4.7 focus on elements of the Linux community, an ecosystem composed of legal persons (**foundations** and **companies** from a variety of IT markets) and natural persons, i.e., **people** (users, developers, coordinators, writers). From GL materials, it is possible to verify that each entity has a particular interest in participating in the community to support and influence the project development.

Foundations

Foundations participate in the Linux project to **support the development community**. To this end, foundations promote related projects and code improvements, produce informative materials, and mediate internal and external conflicts. Foundations also participate in the **project governance** and some of them employ senior developers. Consequently, conflicts of interest became a concern when foundations employ some top-level maintainers and also receive money from corporations. On the one hand, corporation investments provide financial support along with their commercial interest. On the other hand, the top maintainers have voting power in the project and guide the development. Therefore, governance and business are intertwined and this model contributes to keeping the hierarchy and privileges of the upper-cadre.

Authors also raise questions around specific characteristics of foundations' subsistence that would **influence the development and management**. These institutional characteristics include **source of income** (who finances their work), **members** (how its members and board are composed), and **governance** (how they are organized and how they manage the foundation). They approach two prominent foundations in the Linux kernel project: the **Linux Foundation** and the **X.Org Foundation**; however, foundation participation is not restricted to them.

Companies

Companies in Linux development come from a variety of markets and are, occasionally, competitors. They bring **users** and demands o the project. They have different **interests** that vary from providing user support to using the system as a resource for correctness guarantee and becoming more competitive. They contribute to the source code mainly by **employing developers** and their code contributions are identified by the developers' e-mail domain or information on the code submitted.

A company could develop in-house new contributors, but also employ **maintainers** and active contributors that already participate in the process of review and acceptance of code changes. In this sense, employing Linux maintainers could bring some practical facilities for a company, such as clearing maintainer bottlenecks and influencing areas of contributions over a subsystem.

Companies often give some freedom in how employees do the work; however, the work's scope is limited by the company's market interests. Employers are primarily looking for **investment return** and, consequently, pay developers to work on things that bring value to their business. Therefore, despite developer-employees have a large portion of the total code change in each kernel version, the **nucleus of contributions** is around their company's hardware and little in Linux core needs, such as core components and project documentation.

People

"The Linux community is a pyramid" (T. L. Foundation, 2010). The Linux kernel community is hierarchically structured both from the general contributors' perspective and from the code development perspective. People are organized in a **pyramidal structure**, where a large portion of the community of general contributors comprises **users, testers, and sysadmin**. In the mid-level, we find Linux-based **distribution developers** that are, in practice, end users. **Active developers** and **maintainers** are at the top, shaping the system. Maintainers have commit rights and work as leaders, but a large portion of coding is made by developers without commit-rights. They range from the most active to one-time contributors who change the source code by *"submitting patches to maintainers who will, in turn, commit those patches to a repository and push them upstream to higher-level maintainers."* (Edge, 2016).

The hierarchical organization is not new. It is present in different structures that composed the Linux kernel development model and is supported by the git version control tool. In fact, the project management responsibilities looks like a tree. From Linus Torvalds (and from Greg Kroah-Hartman in some contexts), the development coordination starts to branch in maintainers and developers as leaves. Also, the source-code is hierarchical and distributed in many repositories and work trees. Finally, the community roles is organized in levels of participation and also permission rights and privileges that resemble a tree. Hence, the hierarchy models the kernel source, the development process, and also the code development community.

Looking inside the code development community, the **hierarchy is steeper**. The upper-cadre has a top leader with abusive and disrespectful behavior, and the source code is divided into lots and distributed to a small group of maintainers responsible for coordinating work on that slice. In community publications, some individuals claim that the kernel development community is led by a **'benevolent dictator'** – Linus Torvalds. He establishes policy and integrates contributions from all subsystem branches. Strengthening the idea of subordination, a small group of maintainers is below him. For example, in 2014, Jonathan Corbet published an article on the web magazine LWN (Corbet, 2014) starting with *"Any self-respecting development community should be attentive to the needs of its benevolent dictator"*, bringing a firm idea of subservience.

According to publications, git supports this benevolent dictatorship model, encouraging branching and embracing the chaos of allowing developers to code what they are interested in solving. Linus Torvalds is mostly a tie-breaker, and *"the tie-breaker role tends to trickle down to other experienced project members even before a benevolent dictator steps down"* (Edge, 2011).

Interactions

The **means of interactions** can also change between those who are or are not part of the upper-cadre. To meet in person, developers participate in **conferences**. Face-to-face meetings brings **community relation benefits** since members see the faces behind e-mails. Also, conferences help newcomer engagement and sharing experience, ideas, and problem/solutions between different projects. However, some conferences are only addressed to a select group of maintainers, such as the Kernel Summit conference and the Maintainers Summit. These conferences may be seen as private venues to shape the project development, i.e., a **no-public decision-making forum** where discussion, decision-making, and voting take place. , *"A traditional Kernel-Summit agenda item was a slot where Linus Torvalds had the opportunity to discuss the aspects of the development community that he was (or, more often, was not) happy with"* (Corbet, 2017b). The meaning of this slot is questioned in another grey document, arguing that *"feedback sessions about maintainer happiness only reinforce the control structure, with, e.g., the kernel summit featuring an 'Is Linus happy?' session each year."* (Vetter, 2017)

While there are opportunities to meet **in-person**, development community members mainly interact **remotely**. **Mailing-list** is a well-known channel where collaboration takes place. Members from different companies collaborate in **interaction nucleus**, proportional to their companies interest. They are more likely to collaborate with others from the same companies or companies in the same market. Another nucleus is people that recently sent a contribution or were working on the same area of code. Finally, *"someone is more likely to reply to a maintainer"* (Foster, 2017). This corroborates one of the motivations for the multiple-committer maintainership model, the fact that *"Patch submitters wanted to deal with the maintainer rather than with other reviewers"* (Corbet, 2016c).

Discussions occur **asynchronously** per patch and per subsystem. The members are from different companies and **time zone differences** are not an issue. Besides, members keep track of key collaborators' time zones when expecting replies, and similarities in the timezone do not strengthen this interaction. Because it is textual, e-mail interaction can exibit a **lack of tone** and, besides code acceptance, it brings **social aspects** like misunderstandings, hurt feelings, and frustration.

Another means of interactions very present in some subsystems development are **IRC channels**. Because it is also textual, conversations on IRC carries the same social problems of disagreements that could look scary for outside viewers and newcomers. Unlike the mailing-list, IRC interactions suffer from timezone differences.

In fact, social aspects in the Linux community influence the quality of communication. Many community members consider the **discussion culture** positive for the project, where long arguments in flame wars are an essential part of fixing bugs. However, with members distributed geographically and coming from different cultures, communicating

is **context-dependent** and, therefore, members should be careful in the tone adopted to express an opinion.

Developers

Some contribute to the Linux community by spreading the project's practices and recruiting new contributors, such as **book/article authors** and **internship programs coordinators**. However, the kernel developers are those who change the system code.

"The kernel development community is organized as a hierarchy, with developers submitting patches to maintainers who will, in turn, commit those patches to a repository and push them upstream to higher-level maintainers" (Corbet, 2016d). The maintainership model is also evolving as the project grows, and *"the hierarchy shows more clearly than it has in past years. A number of subsystems are growing to the point where there needs to be some overall higher-level coordination. So there are more two and three-level trees than there used to be"* (Corbet, 2017a).

If one day the development of Linux was portrayed as a product of volunteers and enthusiasts looking to solve their personal problems with the system anarchically, this has become a very distant reality. *"Today, over 90 percent of the code is coming from professional developers who are employed by some company to work on the kernel"* (Corbet and Kroah-Hartman, 2017), and professionalism is a trend reinforced by the community.

Identity

Based on Linux community publications, **common characteristics** of a typical development community member is professionalism and discipline. They have diverse backgrounds and are in different locations, working for different companies. On the one hand, Linux developers have a professional and friendly relationship. On the other hand, many do not know each other personally and only "meet" by email or online communication platforms.

Volunteer work is a negative trend in Linux development. The number of contributions from volunteer developers has decreased with each version of the kernel. Active developers report that working on the Linux kernel is a massive career boost. It leads volunteers to be hired quickly, giving them a good **paycheck**. The community claims this is a result of open-source professionalism. Most kernel developers are affiliated with some organization, such as foundations, tech companies, or consulting. However, they do not always link their contributions to the company that pays their salaries. Consequently, in addition to changing the email domain according to the company **affiliation**, they can switch between their personal and company email addresses. Moreover, *"developers who work on independent projects tend to think of themselves as affiliated with the project first, and their employer second; that results in a strong incentive to avoid compromising the project's goals in favor of what today's employer wants"* (Corbet, 2016a).

Motivation to stay

Active kernel developers have different reasons to stay; however, their motivations converge with the Linux kernel project's unique power in the IT market and career opportunities. Developers continue to work on the kernel for **professional** interest, such

as paycheck, career boost, and direct interaction with the system users. They are also in search of **technical inovation** from a cutting-edge operating system: learning new things, getting new ideas, and keeping improvements due to high-level technical collaboration. Finally, but as important as the previous ones, they have **personal motivations** to participate in the community. The variety of Linux usage keeps them never bored – they have fun and meet new people. They also believe their talent is recognized there. And some of them are there for their belief in FLOSS.

Roles

Developers might have a specific role in the Linux kernel community. This role gives to someone priviledges on accepting and committing changes to the source code. In general, the developers should climb a **path of contributions in the community** to acquire a prominent and recognized position. However, the closer to the upper-cadre, the less meritocracy affects – since top-level developers have a reputation, an assumed trust, and generally more coordination than coding activities. The community has an organic influx of **contributors**. These **newcomers** get involved in the development through **internship programs**, **in-house company** trainings, or just **volunteers**. Regardless of the gateway, they deal with a steep learning curve that requires endurance and enjoying the project. From the community members' perspectives, newcomers should have time to learn and synthesize change. Good **start points** are small contributions, such as documentation, patch review, testing, and reporting bugs. For coding contributions, they cited bug fixing as the fastest way to get started. If they start from an internship program (as Outreachy), mentors commonly require knowledge of debugging, git, static code analysis tools, patch submission, driver implementation, and the subsystem structure.

Contributors develop code changes, but only **maintainers** feed the mainline. They maintain a **slice** of the source code that can be a subsystem, distro kernels, stable release, drivers, or a framework. They are hired by companies with a market interest in the slice they maintain. They accumulate a series of **attributions**, from management and quality assurance to training and coding. To feed the upstream with patches, maintainers must gather and manage patches, deal with outstanding issues, as well as review patches. This clerical job needs technical effort and consumes part of their time.

Maintainers should also **concern** about how to lead a team of people, guiding or mentoring developers, coding, and documenting features and procedures. *"It's About the People. If you're maintainer of a project or code area with a bunch of full time contributors (or even a lot of drive-by contributions) then primarily you deal with people. Insisting that you're only a technical leader just means you don't acknowledge what your true role really is"* (Vetter, 2017). Due to their mediator position, maintainers must ensure a balancing act of keeping developers engaged while asking for change or rejecting a contribution. In the meantime, they must prevent failures and regressions in their subsystem, carrying the fear of being publicly censured.

In a subsystem, maintainer adopt a suitable **maintainership model** to sustain daily project activities. Selected documents mentioned three **approaches**: single-maintainer, co-maintainer, and multiple-commiters. A **single-maintainer** works alone, being the only one with commit-rights in the related subsystem repository. Co-maintainer differs from a single one by the number (two). Although most subsystems **adhere** to the single-

maintainership model, the workload on some subsystems requires more people to handle the work. These subsystems already count on proactive developers' feedbacks and patch reviews. To prevent maintainers from burning out, they spread acceptance permission in a **group maintainership model** of two **co-maintainers** or **multiple-committers**. Their **motivation** is to find a robust approach in case of a position vacancy and an effective way to develop new maintainers for the future. Also, as fear is not a bottleneck for changes, this model fosters new contributors and contributions.

Few subsystems have a group of maintainers, but the number is increasing. From publications, we identified three models of group maintainership with the following **general features**:

- *Hands-off:* Maintainers share a single repository and a log file of actitivies. They use an IRC channel to take a "lock" to apply some changes.

- *Delegation:* The work is automatically delegated using the Patchwork patch-management system. Patchwork sorts changes as they arrive and handles each patch to a specific maintainer.

- *Multiple-commiter:* Many committers have the ability to commit changes to the repository; however, maintainers and committers have different responsibilities. While a commiter works internally and is not listed on the MAINTAINERS file; a maintainer has external visibility, dealing with the rest of the world and accepting the blame when something goes wrong.

In both single and co-maintainership models, contributors avoid disagreements with the maintainer since they have only one or two ways to have their contributions accepted. To improve the patch acceptance process, the multiple-committer model emerged in a subsystem with few maintainers and many reviewers. They have observed that code submitters want to deal with maintainers, and this need for a maintainer answer became a bottleneck — only one subsystem reports to use this model. As with the other models, it also presents specific **challenges and requirements** to works correctly. The multiple-commiters model needs a consistent team of developers staying around, and non-maintainer reviews must be the norm to work efficiently. Also, to ensure code quality, committers need to perform useful pre-commit tests and resort to good documentation and tools.

In the maintainer pool, there are also the top-level maintainers, a role filled by **senior developers**. Although most maintainers are employers of a small group of large companies, top-level maintainers are commonly paid by a foundation. They are in singular positions not related to a particular subsystem and almost unavailable for disputes. Their role involves fewer patches and is focused on management and review. Linus Torvalds is at the absolute top and holds the single role to manage the mainline repository and merge pull-requests from all subsystems. *"When asked about group maintainership at the top level, he said that he is open to the idea, but doesn't think that there is a lot of need for it. He manages to be responsive, even when he's off diving in some remote part of the world"* (Corbet, 2017b).

4.3.3 Community Concerns

As expected for a software development project, developers keep an eye out for **technical issues** of code cleanup and standardization, elimination of errors, security/vulnerabilities, and documentation quality. For example, contrary to what is acclaimed by "Linus' law", having enough eyes has not been sufficient for the current security process. *"If we were able to fix problems as quickly as we like to think, it would still be insufficient; it is increasingly clear that patching things up after somebody discloses them is not enough. But, sometimes, problems seemingly slip through the cracks and are not fixed quickly, even when they are known"* (Corbet, 2016b).

However, community publications also bring emergent information and ongoing issues from the social perspective that affect collaboration and the current development model. *"The technical issues have been small compared to the social challenges involved in organizing a project largely consisting of volunteers at first, and then kernel developers paid by companies with competing interests, operating in disparate markets with vastly different computing needs"* (Clark, 2016).

Social challenges

Communication and interaction between members face common social challenges, such as determining the boundary between acceptable and inadmissible conduct, finding mechanisms to identify abusive behaviors, and sensitizing them to cultural differences to prevent disrespectful actions. *"The community likes to think that its decisions are based entirely on merit. In truth, there is also an important social element involved in working in the development community. Despite our efforts to just review code and judge it on its own merits, there are personalities involved. As a result, we see snippy answers and negative things happening in our communication channels"* (Corbet, 2018c).

Conduct issues

Remote interaction and textual **communication in mailing-list** pose challenges to the collaborative development process. Some **negative behavior** is explicit in the use of an **innapropriate language** with profanity and cursing; however, impoliteness and irony are subject to cultural diversity. Participants of the 2018 Maintainers Summit conference (Corbet, 2018b) reported that messages such as *"I love your patch"* from the 0day robot, an automated Linux kernel test service, could be offensive and direct criticism could raise a strong feeling of shame. Maintainers should not attack and be more careful to avoid inappropriate communication.

A common concern reported by participants and observers is related to the governance model of the Linux kernel, which gives Linus Torvalds enough power to sustain his **offensive behavior** (Brodkin, 2013). Selected documents describe his rude and aggressive reputation, and some members report that his controversial behavior has hurt and drove away developers. Nevertheless, **active-developer resigners** do not restrict the "toxic" working environment to him, reporting it in various high-profile people's behavior. Unsatisfied with the development process or community norms on mailing-lists, few make noise when deciding to step down.

The conduct of community members is an **ongoing issue**. The longstanding **Code of Conflicts** was no longer efficient to prevent negative behaviors. Even worse, the Code of Conflicts acculturated *"no-filter feedback and bluntness as the natural and more successful state of open source software development"* (Statt, 2018). During the term of the Code of Conflicts, only three conduct issues were reported to the **Linux Foundation Technical Advisory**, and all of them had weak substance.

After a self-imposed short break from the Linux development (Torvalds, 2018), Linus Torvalds replaced the Code of Conflict with a **Code of Conduct**[9] in an attempt to avoid losing talents and an answer to a magazine article questioning his conduct on public mailing-lists (Cohen, 2018). The Code of Conduct was presented as a **symbol of the community conduct goal** to continue to scale, remain a welcome place, and convince the outside world that the community has gotten better. It is a consequence of the progress toward civility and professionalism in the Linux kernel community.

The code was **based on Contributor Covenant Code of Conduct**[10], which other projects are used but is not widely accepted by the community. Its adoption was inspired by the DRM subsystem's successful case of conduct, based on the freedesktop.org code. The code was reviewed by Lawyers of the Linux Foundation and, in case of conduct incident, someone should report to a Code of Conduct Committee, composed of volunteer members of the Linux Foundation Technical Advisory Board and a professional mediator.

Despite impacting all community members, the code was initially **built in private**, without a community-wide discussion. Members complained that the **adoption process** was hasty and did not follow the open-source model, and the acceptance does not happen in a public manner but through hidden e-mails. Also, when published, it was first presented for a group of maintainers. This group raised several questions about the application of norms and the maintainer's responsibilities on that. Linus Torvalds clarified that maintainers are not police and should lead by example, and the group asked for additional interpretation documents[11].

Again, questions about the Code of Conduct present in some publications reveal how interaction problems in the Linux kernel community are similar to the other social groups. *"Such laundry lists of misbehavior can leave a bad taste in the mouth; that can be especially true if, like me, you are an older, run-of-the-mill white male; it is easy to look at a list like that and say everybody is protected except me'. It can look, rightly or wrongly, like a threatening change pushed by people with a hostile agenda"* (Corbet, 2018a).

Process Issues and Challenges

In 1998, Linus Torvalds merged some incomplete frame buffer patches to the 2.1.123 release, which caused a compilation failure in the Linux kernel. That served as a trigger for some members disappointed with the Torvalds' manner to handle patches. This episode became known as the "Linus burnout" and shed light on **scalability issues** in the Linux development model (Corbet, 2010). The Linux kernel project has been growing steadily.

[9] https://www.kernel.org/doc/html/latest/process/code-of-conduct.html
[10] https://www.contributor-covenant.org/
[11] https://www.kernel.org/doc/html/latest/process/code-of-conduct-interpretation.html

Since all contributions' endpoint is the mainline repository, and the version's release is centered on Linus Torvalds, the "Linus not scales" problem still permeates the community discussions.

"Process scalability requires a distributed, hierarchical development model" (Corbet and Kroah-Hartman, 2017). Because of this, the Linux development model has undergone some transformations supported, basically, by the adoption of version control tools – Bitkeeper and later Git – and by the hierarchical structure of maintainers. These smoothed the release process dependence on Linus availability and increased the volume of patches he could handle. *"Without the right tools, a project like a kernel would simply be unable to function without collapsing under its own weight"* (Corbet and Kroah-Hartman, 2017).

Now, other nodes in the development process seem saturated. Through publications, the community points out the challenges in the **maintainership scalability** and the patch acceptance and **review process**. In addition to tools to support the release cycle process, the community claims for more documentation and automation of the contribution flow daily tasks, expressing concerns with the project's continuity as the community is **getting older**. The community publications present substantial guidelines, supporting tools and online materials, and **internship programs** to **attract new and young people**. However, after three decades, they are now facing problems of a matured, long-life project. The upper-cadre is aging, and maintainers are also not self-motivated to find people able to substitute them in case of vacancy.

Hence, in spite of the interest in attracting outside developers, the community currently turns part of its attention to find mechanisms able to retain and **grow the existing contributors**. It needs contributors to replace or scale roles important for the project workflow, like maintainers and reviewers. The capacity of reviews must grow, and maintainers show **signs of stress**; therefore, developers other than the maintainer should be available and included in the review process. More reviewers, as well as **mid-level maintainers**, would reduce the maintainership workload. Many maintainers have limited time working on a given subsystem, and the situation worsens as the subsystem grows in size. A non-proportional number of developers can overwhelm their work, leading them to look for another job. A burnt-out maintainer is also more likely to ignore or reject new ideas and give short and unclear answers.

The concern around scaling the number of maintainers increases as the number of subsystems is growing. This growth requires higher-level coordination, two and three-level trees, and, consequently, the progression of developers in maintainers. The community claims for documentation and automation to avoid **knowledge retention** and make more friendly and **inclusive processes** to both new contributors and developers already part of the community.

Standardizing the process can also avoid the **patch acceptance double standard**. The probability of merging a patch increases from newcomers to the long-term contributor and contributor commits to maintainers self-commit. Maintainers have implicit assumptions and rules and use checking and testing tools unavailable to developers. Some maintainers **resist changes in the development process** because they fear losing privileges. By retaining knowledge and requesting proof of subordination from those who need the

approval to move on, they turn the patch acceptance process into a superior power exercise that complements their toxic behavior.

The results obtained from the strategies selected for the grey literature review on the Linux kernel project reinforce the relevance of examining FLOSS community publications. Grey literature is a rich source of data to define key elements and accurately describe the subjects of a FLOSS project development model. Besides mapping roles, rules, structures, and mechanisms of the distributed, community-based Linux kernel development, our investigation reveals current community concerns that can affect the project's survival. Therefore, we advanced our investigation to assess the coverage and pace of academic research and day-to-day development issues in the Linux kernel project. In the next chapter, we present the results of systematically review academic studies, the evaluation of adherence to academic and grey literature concepts, and discussion findings from a participant observation in the Linux kernel community.

Chapter 5

Different perspectives on the Linux Kernel Development Model

This chapter presents an up-to-date comprehensive description of the Linux kernel community from different perspectives of FLOSS community publications, Software Engineering studies, and Participant Observation. We compare concepts from grey to traditional literatures in the last decade, and highlight convergence and divergences on undestanding the current Linux kernel community model. We directly participated in the development community routine to discuss our findings and shed light on pitfalls in mistaken assumptions.

5.1 Multivocal Literature Review

In this research, we conducted a multivocal literature review to embrace the concepts from the current Linux kernel development community theory and practice. Multivocal literature can summarize both state-of-the-art and -practice. The comparison between its findings also is fruitful to identify gaps and conflicts between theory and practice in the comprehension of the FLOSS phenomenon and its transformation, improving the conduction and relevance of FLOSS research in software engineering.

In conducting a multivocal review, a researcher needs to consider the documents' differences of structure standards, the rigor of the review, and how their findings are supported. Due to the FLOSS continuous innovation characteristic, software engineering researchers and practitioners may also not be at the same pace. These incompatibilities can result in studies with outdated information and also misunderstandings.

To overcome these diverse challenges, we opted for a research strategy based on findings from GLR and subsequent correspondence with SLR codes. Therefore, we reviewed the grey literature on FLOSS to find the Linux kernel development community's state-of-practice and its convergence with the bazaar model, i.e., the open-source paradigm (Sanders, 1998). We conducted a systematic literature review and, for mapping congruences and incongruities, we analyzed SLR data and compared their codes based on the GLR findings.

5.1.1 Systematic Literature Review

This research phase aims to compare, using a systematic literature review, the attributes used by Software Engineering studies to characterize the current Linux development community model with those systematized by the previous Grey Literature Review.

The primary studies were identified by querying digital libraries and following a methodical protocol for study selection. Using a directed approach to content analysis, we compare and extend conceptually the framework obtained from the GL. The content analysis also exposed a certain lack of pace between what is investigated in academic studies and the Linux kernel community concerns.

Studies Selection

We design a systematic literature review protocol (Appendix B.1) based on recommendations from Almeida Biolchini et al. (2007), Barbara Kitchenham, Pearl Brereton, et al. (2009), and Higgins and Green (2008). From the problem outline that guides this research (Section 1.1), the systematic review aims to answer the software engineering studies perspectives of *RQ1. How do software engineering studies and Linux community publications describe the current Linux development community model?*

We selected documents from journals and peer-reviewed conferences available in the following digital libraries:

- IEEExplore (https://ieeexplore.ieee.or)
- ACM Digital library (https://dl.acm.org/)
- ScienceDirect/Elsevier (www.sciencedirect.com)
- Springer (https://link.springer.com/)
- Google scholar (scholar.google.com) [For double checking]

Traditional publishers and their standards for publication provide helpful search engine tools. On the other hand, the Linux kernel is used in various markets, and the system and its development project have characteristics relevant to a wide range of interests in software engineering studies. To mitigate inconsistency issues in studies of Linux and increase the search-string efficiency, we tested the accuracy of the same terms used in the GL search process by performing several search iterations until reaching a search string with synonyms and composed terms:

"Linux kernel" AND ("development model" OR "community practices" OR "development practices" OR "community rules" OR "development process")

As opposed to grey literature databases, digital libraries have a higher quality of search engine and homogeneity in features and listing of search hits. Consequently, we obtained a more efficient result from defining a search string with composed terms, i.e., words combination. We considered the same time interval of GLR to retrieve traditional publications (from 2009 to 2019+). We applied the query mentioned above in four databases in March 2020, as shown in Table 5.1, and obtained a total of 1081 candidate-documents.

Table 5.1: *Search string results per digital library*

Database	# of hits	# of duplicated
IEEExplore	96	3
ACM Digital Library	335	27
Science Direct	203	5
Springer	447	6

Table 5.1 summarized the number of documents obtained from each digital library. These repositories enable us to filter only online avaliable documents written in English; therefore, we had no effort to evaluate the compliance of this first two eligibility criteria, as defined in our SLR protocol, in Appendix B.1. Whenever possible, we downloaded the reference hits as CSV or Bibtex files. We stored the references lists on the Mendeley tool to facilitate metadata evaluation. After, we transposed the gathered references to an online spreadsheet on Google for documenting the entire screening process. Using the Mendeley's feature of "Check for Duplicates", we found 41 duplicated documents, marking them as duplicated in the spreadsheet for exclusion.

Identification and Screening Process

Before starting the screening process, we classified the publication venue according to its context relevance for software engineering and FLOSS studies, as shown in Table 5.2.

Table 5.2: *First step - Evaluating publication*

Publication level	Meaning	Criteria
-1	Excluded: Out of context	Publication venue and Publication keywords
0	Excluded: Out of interest	Book chapter and Less than 10 pages
2	Less Priority	Magazines
3	High Priority	Journals and Conferences paper with more than 10 pages or focused on FLOSS
*	Unclear	

From this classification, we identified publications in FLOSS and Software Engineering topics more likely to meet our eligibility criteria with stronger findings. We excluded 677 documents in this classification step. To the next step of screening, we selected 363 documents with more than 10 pages published in journal and conferences on FLOSS and SE research venue.

Reviewer 1, the leading researcher of this work, performed the initial phase of the screening process to exclude documents by title that clearly meets at least one exclusion criteria. She excluded 173 documents because their title are clearly out of the context of this research. Even though Higgins and Green (2008) recommend the participation of two or more reviewers in the screening process, a single reviewer in the initial step is acceptable.

At this point, we conducted a participant-observation in the Linux kernel project, and Reviewer 1 became a member of the project development community. We also introduced

a second reviewer for the first round of text evaluation. Here, we identified this reviewer as Reviewer 3 since this one is a different person than Reviewer 2, who participated in the GLR screening process. Different from Reviewer 2, Reviewer 3 is not a Linux developer. Reviewer 3 is a Software Engineering researcher with knowledge in sound FLOSS and Software Engineering practices.

From a reduced sample of 190 documents, Reviewer 1 and 3 worked independently to evaluate title and abstracts by eligibility criteria. At the end of the process, Reviewer 1 and Reviewer 3 compared their selection samples and found 31 conflicts. After discussing conflicts, they agreed to exclude eight from 31 conflicting decisions, converging on the exclusion of 129 studies by title and abstract evaluation. Therefore, in the last step, we read the full text and evaluated 61 documents, excluding 36 documents that do not meet the eligibility criteria. Table 5.3 present the total of documents excluded by each exclusion criteria.

Table 5.3: *Results from each identification and screening phases*

Exclusion criteria	Total
Duplicated	41
Scope of Publication	677
Title out of context: Linux kernel project or FLOSS is not subject of study	173
Restricted to technical issues	14
Linux kernel is not subject of study	66
Restricted to validate a method, framework or tool	22
Linux kernel development practices is not subject of study	63

From this process, we selected 25 documents for content analysis. Table 5.4 shows the distribution per database of documents retrieved by each search engine and documents selection after the screening phase.

Table 5.4: *Selection process per database*

Database	# of hits	# selected studies
IEEExplore	96	2
ACM Digital Library	335	9
Science Direct	203	5
Springer	447	9

The entire selection process is documented in spreadsheets[1]. Each column corresponding to a screening phase, where we detail reasons for exclusion. Whenever we could assess that a single eligibility criterion failed, we excluded the document: marked as excluded in the respective phase (column) and reported this criterion.

As summarized in Table 5.5, we identified the data source considered for investigation, whether a study includes or not grey literature, and how this literature is used in each

[1]Data available on https://gitlab.com/ccsl-usp/glr/-/blob/master/SLR_Linux_kernel_development_model.ods

Identifier and Reference	The use of Grey Literature	Multicase	Artifacts	Interview
SE1 Bagherzadeh et al. (2018)	Background and Research Design	No	Source code and Commits	No
SE2 Palix et al. (2014)	Introductory	No	Source code and Commits	No
SE3 Tan and Zhou (2019)	Analysis of Online Documents	No	Patches	No
SE4 Tian et al. (2012)	Background	No	Commits	No
SE5 Zhou et al. (2017)	Analysis of Online Documents	No	Source code	Yes
SE6 Rigby et al. (2014)	Shallow	Yes	Patches	Yes
SE7 Izquierdo and Cabot (2018)	Analysis of Websites	Yes	-	No
SE8 Jiang et al. (2014)	Introductory	No	Patches and Commits	No
SE9 Izquierdo-Cortazar et al. (2017)	Background	Yes	Patches and Commits	No
SE10 Zaidenberg and Khen (2015)	Introductory	No	Source Code	No
SE11 Lindberg et al. (2014)	Analysis of Websites	Yes	-	No
SE12 Shaikh and Henfridsson (2017)	No	No	E-mail	No
SE13 Avelino et al. (2018)	Background and Discussion	Yes	Commits	No
SE14 Feitelson (2012)	Analysis Online Documents	No	Source code	No
SE15 Israeli and Feitelson (2010)	Discussion	No	Source code	No
SE16 Berger et al. (2014)	Analysis of Online Documents	Yes	Source code	No
SE17 Riehle and Berschneider (2012)	Analysis of Websites	Yes	-	Yes
SE18 Barr et al. (2012)	Analysis of Online Documents	Yes	Source code and Commits	Yes
SE19 Lotufo et al. (2010)	Background	No	Source code and Commits	No
SE20 Forrest et al. (2012)	Background and Discussion	Yes	E-mail	No
SE21 Eyolfson et al. (2014)	Shallow	Yes	Commits	No
SE22 Capiluppi and Izquierdo-Cortázar (2013)	No	No	Commits	No
SE23 Joblin et al. (2017)	No	Yes	Commits	No
SE24 Bettenburg et al. (2015)	Analysis of Online Documents	Yes	Source code and E-mail	No
SE25 German et al. (2016)	Background	No	Commits	No

Table 5.5: *Selected papers*

selected study. Twelve studies use the Linux kernel project as the unique case of study, and the remaining thirteen analyze and compare the Linux kernel with other FLOSS projects. Most primary studies resort to grey materials to obtain some background to conduct their investigation and discuss findings. Six from 25 selected papers include the analysis of grey documents as sources of evidence, but none of them collected these documents with a systematic approach.

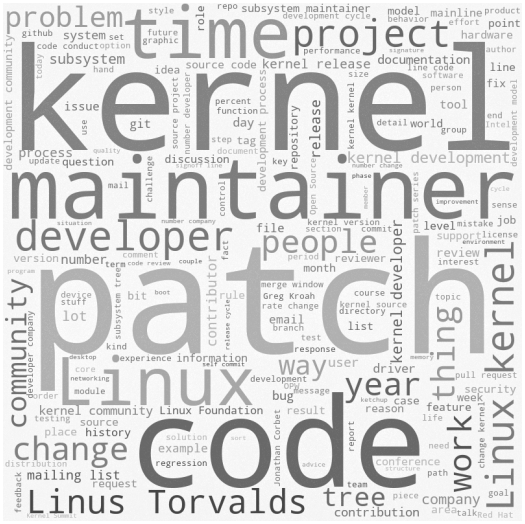
We observed that most selected studies resort to grey literature to develop their investigation background, commonly in a non-systematic manner and shallow analysis of data. Bagherzadeh et al. (2018) massively used GL to support the investigation background and research design, but do not use these documents for data collection. Zhou et al. (2017) inspect Linux websites looking for project-related information and examine various blogs, forums, webzines, and news websites to understand maintainer behavior and design suitable measures for maintainers' work. However, the paper presents shallow information on their approach to collect and analyze this kind of data. Tan and Zhou (2019) collected and analyzed online documents to understand communication when submitting patches in the Linux kernel development process. They use an exploratory approach, searching two keywords on Google search, gathering the first 50 hits and, from them, snowballing documents.

All selected studies consider at least one type of project artifacts. Most of them collected

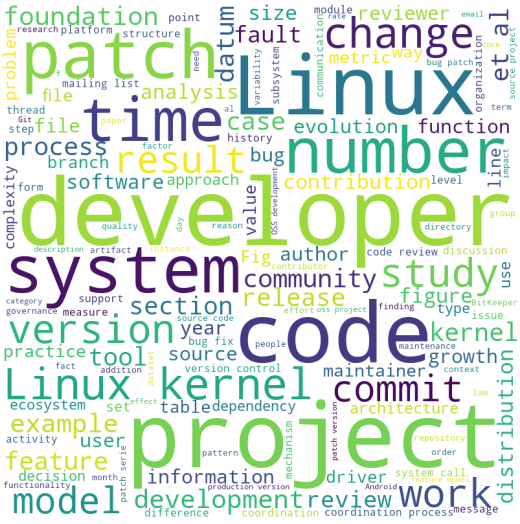
data from source code repositories (18 studies), which could include source code (10 studies) or commit messages (12 studies). Seven of them analyzed data from mailing list archives. Finally, four studies used project profiles and websites to collect data. Ten studies used more than one data source in their analysis. Besides project artifacts and public project documents, other sources of data used by primary studies include questionnaires and interviews. None of them systematically analyzed data from grey literature. However, we verified in the sample of primary studies GL references that matched GL documents selected in our previous phase of GLR.

5.1.2 Content Analysis

We followed a directed approach of content analysis (Hsieh and Shannon, 2005) to compare labels in the mind map from the GLR with the content present in academic publications for the same period and similar eligibility criteria.



(a) *Grey Literature Review- Word Cloud (Figure 4.2)*



(b) *Systematic Literature Review - Word Cloud*

For comparison with GLR keywords, we generated a word cloud from the text data of all documents selected from SLR, as presented by Figure 5.1b. The word cloud enabled us to promptly note convergences and divergences of interest between these two sources of evidence. Both discuss key concepts of the project development model. They present keywords that describe the basis of Linux development, such as time, community, mailing list, developers. Other words describe project artifacts, such as code, patch/change, source project, and repository.

On the one hand, the grey literature focuses on community subjects, such as people, maintainers, discussion, conferences, code of conduct, proper names of well-known profiles, and emphasizes the subsystem organization. On the other hand, and as expected, the formal Software Engineering literature shows keywords regarding process subjects, such as coordination, metric, decision, organization, communication, size, author, and fault. Consequently, comparing both states of mapped concepts strengthened our previous decision to focus on the community model that supports the Linux kernel development.

We started the directed content analysis by highlighting all passages in the text that

appears to be related to aspects of the Linux kernel development community. We coded these passages considering codes predetermined by GL and gave a new code to the text fragment that could not be categorized using the initial GL coding scheme. Hence, we could support findings from the previous phase and determine the relationship between each keyword's theory and practice.

Researchers are more likely to find evidence supporting the preexistent theory by using directed content analysis (Hsieh and Shannon, 2005). However, in our experience from the grey literature to the formal ones, we identified gaps and mapped divergences of understanding related to the project development and primary concerns. Moreover, by opting for grey literature first, our researchers were not blinded to social and contextual aspects that mobilize the community to structure and restructure the project development process and organization.

5.2 Unifying Academic and Community Understandings

This phase's primary goal is to identify which variables of interest in the Linux kernel community publications were already explored or are under investigation by Software Engineering studies. We also intend to understand how they were addressed to identify which of them potentially need further investigation. Therefore, we obtained the codes by directed content analysis of the primary studies. We used the complete version of the grey literature mind map[2] as the initial code scheme. This version includes attributes and categories emerged by reviewing community publications. Whenever the same code occurrence is found, we evaluated whether the study's topics go through the entire branch or has an unclear base or shallow approach.

We mapped all codes from our SLR in the GLR mind map presented in Chapter 4. This mapping describes how well topics in the primary studies fit the selected community publications. We present the comparative map divided according to the subjects in Figures 5.2, 5.3, 5.4, 5.5. The comparative mind map highlights three levels of Software Engineering studies coverage regarding FLOSS community topics: unclear (no-mention), some discussion but shallow analysis, or addressed (investigated with findings).

In the following subsections, we relate the concepts by analyzing 108 publications from the Linux kernel community with those present in 25 academic studies selected in this systematic literature review. Due to the figures' size, a better understanding of the following section requires the reader to consult mentioned figures while reading. As an alternative the entire mind map is available online in a git respository[3].

[2]Mindmap available on https://gitlab.com/ccsl-usp/glr/-/blob/master/Linux_KernelDevelopment_Comunity_Full.pdf

[3]Comparive mindmap available on https://gitlab.com/ccsl-usp/glr/-/blob/master/Linux_KernelDevelopment_Comunity_Categories.pdf

5.2.1 General Characteristics of the Linux kernel Community

In community publications, authors resort to analogies to describe the Linux kernel development structure. Despite subjectivity, we can relate the structures and meanings of an analogy in similar definitions, but more formal and standardized in FLOSS academic studies. Therefore, in Figure 5.2, we compare grey and academic literature on topics of interest regarding the Linux kernel development community's general characteristics.

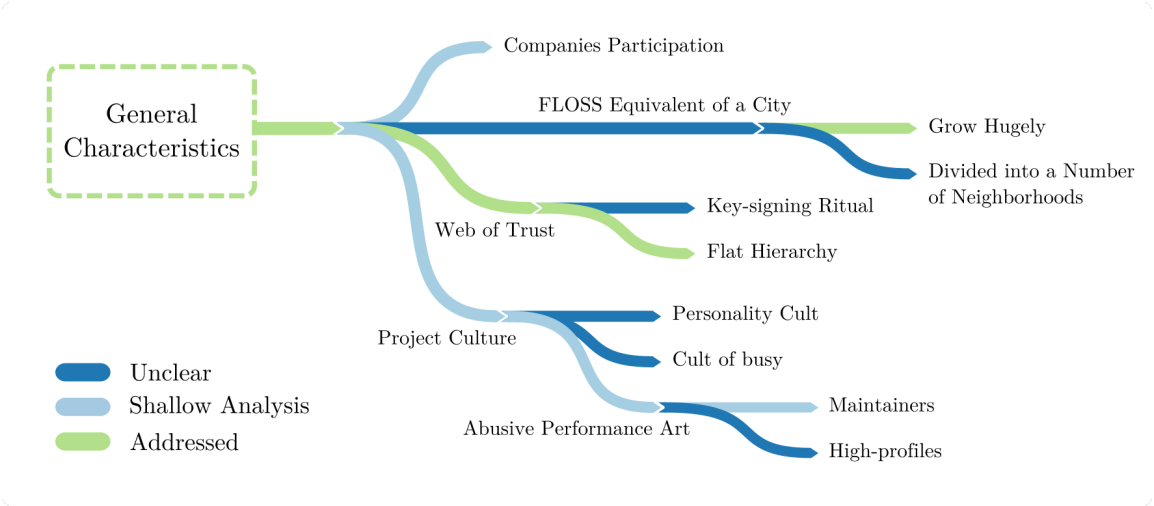


Figure 5.2: *Comparing interests of academia and community publications – General Characteristics*

We observed that studies have explored the chain of trust in the maintainers' tree and between maintainers and developers (Rigby et al., 2014; German et al., 2016; Zhou et al., 2017). Shaikh and Henfridsson (2017) exposes how chain-of-trust mechanisms are adopted to reinforce authority and privileges in the Linux development community. *"Technology is often embedded with rules and structures that are seen as part of good design and requirements, yet in practice, users are less able to notice such rules unless they face them as a possible restriction in use. Oligarchic recursion established the powerful actors and further entrenched them into the Linux project through instituting nested and recursive practices for code approval, shortlisting of patches or developers for different roles, or the choice of new maintainers."* German et al. (2016) describe the patch flow into the mainline by commits propagation via pull requests. They focus on the history traceability across distributed repositories and state that *"most repositories are controlled by one person who has the ultimate power to decide what makes it into that repository"*.

On the other hand, no primary study has addressed the use of signature keys on pull requests as an identity verification mechanism for security. Also, studies present few attributes to describe social aspects and characterize community organization's overview. Finally, discussions of the impacts of project culture on members' interaction and development collaboration between subsystems are shallow.

5.2.2 The Community Ecosystem

The entities that compose the Linux kernel community are sources of interest for academic studies and community publications. In general, we observed a good coverage and convergence in these topics comparing both grey and formal bibliographic literature, as shown in Figure 5.3.

Recent studies have investigated the project structure to support coordination and governance. We found studies on the Linux kernel project that analyze and compare their findings with some particular characteristic of a bazaar-like model. Lindberg et al. (2014) claim the Linux kernel project is an example of a cathedral-like development model, with an oligarchic social distribution *"where important knowledge is controlled and shared by several core developers, but where knowledge and control are highly concentrated and only part of them extend to the entire community"*. Despite knowledge restriction, the structural kernel distribution has a high level of interpretative flexibility since the community has *"more freedom in how they combine and utilize heterogeneous artifacts while developing OSS projects"*.

On the other hand, Joblin et al. (2017) investigate developer-coordination on 18 large FLOSS projects, including the Linux kernel, and found a hybrid coordination structure between core and peripheral groups of developers. Shaikh and Henfridsson (2017) present even more diversified characteristics for the Linux governance. They gathered data from the *Linux kernel Mailing List (LKML)* to examine the coordination process along with the project history until the adoption of git (2005). As a result, they described project governance under continuous transformation. The community sampled different version control tools to support the dramatic and sustained growth in numbers of developers and contributions. Along with the project history, different combinations of four management processes co-existed: autocratic clearing, oligarchic recursion; federated self-governance; and meritocratic idea-testing. For them, this multiplicity of authority structures embraces the diverse motivations of different sorts of developers in large-scale, distributed FLOSS projects such as the Linux kernel.

Software Engineering studies have explored project artifacts on version control tools and mailing lists as data sources regarding coordination and communication. Avelino et al. (2018) describe the nucleus of collaboration and discussion between Linux kernel authors by analyzing the authorship across files of 66 stable releases. German et al. (2016) analyze how Linux uses git and emphasize that *"git provides the freedom to participate in the development, but git does not provide the freedom to contribute."*. To Shaikh and Henfridsson (2017), *"the distributed nature of collaborating entails a greater dependence on such coordination tools such as email and version control software"*.

In fact, the Linux kernel development is mostly e-mail driven, where contributions and reviews take place in patches sent to mailing-lists. Tan and Zhou (2019) map accepted and rejected patches sent to *LKML* to obtain sound practices for communication when submitting patches. The mailing list is a common data source for analyzing the review process. Rigby et al. (2014) identify patch contributions on the mailing list by examining threads looking for diffs. They consider patches that received a response to analyzing the review processes of the Linux kernel and 24 FLOSS projects. Jiang et al. (2014) perform an empirical study on patches sent to the Linux kernel mailing list and

trace the related commits to characterize the reviewing history in a low-tech reviewing environment. Izquierdo-Cortazar et al. (2017) analyze and propose a methodology to study Linux-style code review. They say, in the Linux-like development model, developers from many different companies collaborate to maintain a common code base. In this sense, Forrest et al. (2012) explore the participation of outside organizations in the FLOSS project. They selected the Linux kernel and GCC projects because they use complete e-mail addresses in Bugzilla and code repositories and had open and widely available mailing list archives.

As mapped from GL, outside organizations are part of the Linux community ecosystem. Riehle and Berschneider (2012) analyze the structure and processes of Linux Foundation and eight other FLOSS foundations. They model foundations categories in terms of general characteristics, philosophy, intellectual property, governance, financing, and operations. Izquierdo and Cabot (2018) analyze the nature of 89 software foundations, and characterize more than 60 FLOSS Foundations to describe the role of these foundations in FLOSS projects. Among the Linux kernel-related, they selected the X.Org Foundation to study their openness and influence in the development practices. However, they did not focus on the Linux Foundation (LF) because it did not obey their eligibility criteria of independence. They assess the LF *"follows a flexible approach which serves as an umbrella for its projects, which can deploy specific development processes, and therefore concentrates more on the promotion of OSS benefits."*

Forrest et al. (2012) use different metrics tracking participation and influence in projects to analyze whether businesses and other organizations are biased in their participation. They search for evidence of participation bias in terms of bug reporting and code contributions and claim that *"This knowledge is not just important to the projects themselves, but to potential FOSS adopters or developers. Understanding who supports and influences the project is crucial to making better decisions about whether this is a project worth investing in."* Although they stated that *"The responsibility for managing FOSS projects is in the hands of project maintainers. These individuals manage the code; they are responsible for choosing which contributions to incorporate into a release, and who has the ability to submit code"*, none of the selected studies explore the influence of the foundations and companies' interests in the level of maintainership and project governance, as inferred by GL documents.

Finally, members' communication still deserves further examination. The lack of good social communication skills between members may spill over the collaborative work that supports a distributed, community-based development and negatively impacts the community's sustainability and growth. Capiluppi and Izquierdo-Cortázar (2013) analyze commit-date and timestamp on git to assess developers' activity patterns, and Eyolfson et al. (2014) address developers' timezone to investigate code quality and the propensity of bug introduction. None of the primary studies examine how timezone differences may affect collaborative work and address the nucleus of collaboration in different communication means. Besides mailing-lists, community interactions occur in more informal and instantaneous venues, such as IRC and in-person meetings as conferences. Tan and Zhou (2019) state that *"for patch submission in the Linux kernel, the medium is fixed: email. Thus, it is important to explore how a message is expressed and what the context is"*. However, our findings on GLR show that other communication venues are

a good source of the project decision-making process in the daily maintainership tasks of patches review and acceptance and a broad perspective of governance.

The GLR briefly cites individuals' profiles unrelated to the code's contributions: internship coordinators, writers, users, and testers. Although the selected studies address only users and testers, German et al. (2016) uncover activities that were before invisible and detail three active end-users roles. Their contributions do not involve code commits into the mainline: integration testers, experimenters, product-line maintainers. They also divide code contributors into two roles concerning mainline contributions: producers and integrators.

Figure 5.4 continues to describe community profiles, focusing on code developers. In the hands-on perspective of coding and review, Software Engineering studies and community publications use common characteristics to describe Linux kernel developers. Avelino et al. (2018) use the e-mail address to identify unique developers and Forrest et al. (2012) mapping e-mail domains to assess contributions regarding contributors affiliation.

Community publications usually focus on long-term members, addressing what motivates a developer to stay, what kind of motivation led them to join the project, and whether their interest was satisfied or if their interest has undergone changes. On the other hand, software engineering studies usually investigate the motivations and barriers of less experienced kernel developers. Moreover, these studies tend to view new joiners as volunteers with personal interests and who need to deal with inclusion barriers independently. For Forrest et al. (2012), *"Large projects may not be accurately portrayed as grass-roots volunteer efforts."* Therefore, further studies may investigate why developers decide to stay in the community or drop out and whether barriers and motivations could change depending on how this new contributor was introduced in the community and its affiliation.

Finally, few studies examined the maintainership structure or the coexistence of different maintainership models in the Linux kernel project. Zhou et al. (2017) already claimed that *"Even though a substantial body of literature has characterized developers' work, what maintainers do has not been addressed"*. Their initiative to understand maintainership scalability analyzes data from software engineering literature, online documents, interviews, and mainline repository to quantify maintainers' work. Their study slightly reports different maintainership structures in the Linux kernel; however, no primary study explores those models.

5.2.3 The Community Concerns

Recent studies have already evaluated some current concerns in the Linux kernel review process. The community is looking for mechanisms to grow the review capacity and encourage developers' code reviews other than maintainers. Bettenburg et al. (2015) shows that peer review in the Linux kernel is organized less formally and follows a hierarchical approach. These characteristics affect the transparency and documentation of the decision-making process. Because of the voluntary management based on e-mails, the first feedback usually takes a significant amount of time to happen. Moreover, contributions are either abandoned, rejected, revised, or accepted.

Rigby et al. (2014) state that *"earlier research into the optimal number of inspectors has indicated that two reviewers perform as well as a larger group"*, in contrast to the Linus' law. Palix et al. (2014) also weigh the accuracy of the law, since *"in practice, code that is frequently executed, or at least frequently compiled, is more likely to be reviewed than the rest. When code is frequently executed, many users are likely to encounter any faults and some may fix the faults themselves or submit a request that the faults be fixed by a kernel maintainer. When code is frequently compiled, even if it is not frequently executed, it can easily be submitted to fault-finding tools that are integrated with the kernel compilation process."* Izquierdo-Cortazar et al. (2017) shows the impact of the number of reviewers in the time-to-merge metric for a Linux-like code review process. They also point out the community should overcome the challenges of a low-tech review environment and many e-mail messages to improve the project review process.

Also related to code review, the primary studies approach the double-standard in patch acceptance. Palix et al. (2014) show that committers (maintainers) generally present lower faulty commits proportionally for all their commits, but their faulty patches are still applied in the repository. Tan and Zhou (2019) reveals that patches sent by maintainers have more chance of acceptance, even with poor commit description and explanation, because reviewers tend to trust and predict quality. They also found that reviewers are stricter with cleanup patches than with bug fixes and more likely to accept changes previously discussed or a community need. Bettenburg et al. (2015) observed a statistically significant acceptance bias towards more substantial contributions. Jiang et al. (2014) states that *"Having to submit additional patch versions is not a disaster, since relatively more such patches are accepted than for patches with just one version, and reviewers keep on being interested in subsequent versions."* However, in some community publications, authors suggest a limit between real improvements and proof of subordination in repeated rejections. Also, for Shaikh and Henfridsson (2017) and Lindberg et al. (2014), the processes of code review and patch acceptance in the Linux kernel are mechanisms for decision-making that strengthen privileges to the small group of kernel maintainers.

Avelino et al. (2018) discuss how knowledge restriction affects the recruitment process for core and architectural parts of the Linux kernel system, as *"developers must have great confidence on the changes they propose, discouraging volunteers from performing small changes as a means to become kernel contributors."* Zhou et al. (2017) also state that core modules require a higher skill level and people specialized and more experienced. In contrast, implementation in peripheral modules is relatively self-contained, providing a lower entry barrier for newcomers. Also, periphery parts keep growing because of hardware manufacturers' interest, where commercial entities add code to the driver's module and leave the community to resolve conflicts in core modules. Consequently, their studies found less development activity in core modules and stable maintainer's workload, while both continue to grow in the periphery.

Bagherzadeh et al. (2018) do not address much about community. On the other hand, they discuss how to improve the community development process in the system calls field with knowledge sharing via documentations and automated testing and refactoring tools. Zaidenberg and Khen (2015) also claim that existing tools are not sufficient to detect kernel security vulnerabilities and present a framework for kernel profiling, code coverage, and simulations during the development, similarly to a debugger. Tian et al. (2012) propose

an approach to automate bug-fixing patches identification to support maintainers of long-term stable versions and Palix et al. (2014) claim for the relevance of fault-finding tool and checkers to ensure code quality.

From Figure 5.5, we can quickly realize that software engineering studies address few topics that currently concern the community about its development model. Our multivocal literature review suggests shallow investigation in social challenges of members' conduct issues in a large-scale, distributed community of developers. Also, the selected papers do not examine issues related to active developers with less external notoriety. Evaluating the inclusion of mid-level maintainers instead of more maintainers in the same level may address scalability for maintenance and review processes. Also, we could not find power assessment for internship programs and mechanisms to grow developers into maintainers and make them able to replace upper-cadre members, or if there are any non-explicit barriers to this.

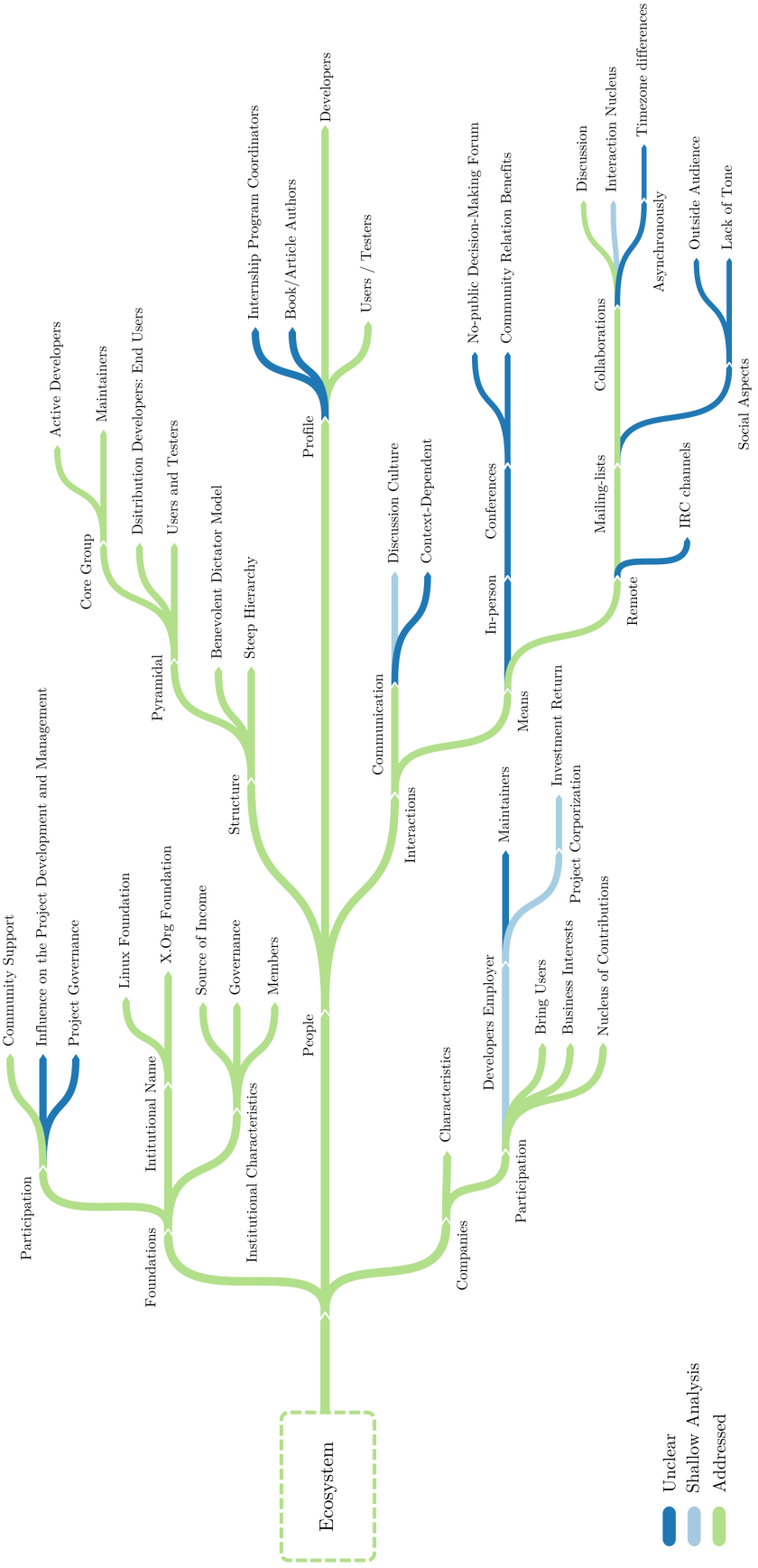


Figure 5.3: *Comparing interests of academia and community publications – Attributes regarding the Community Ecosystem (1)*

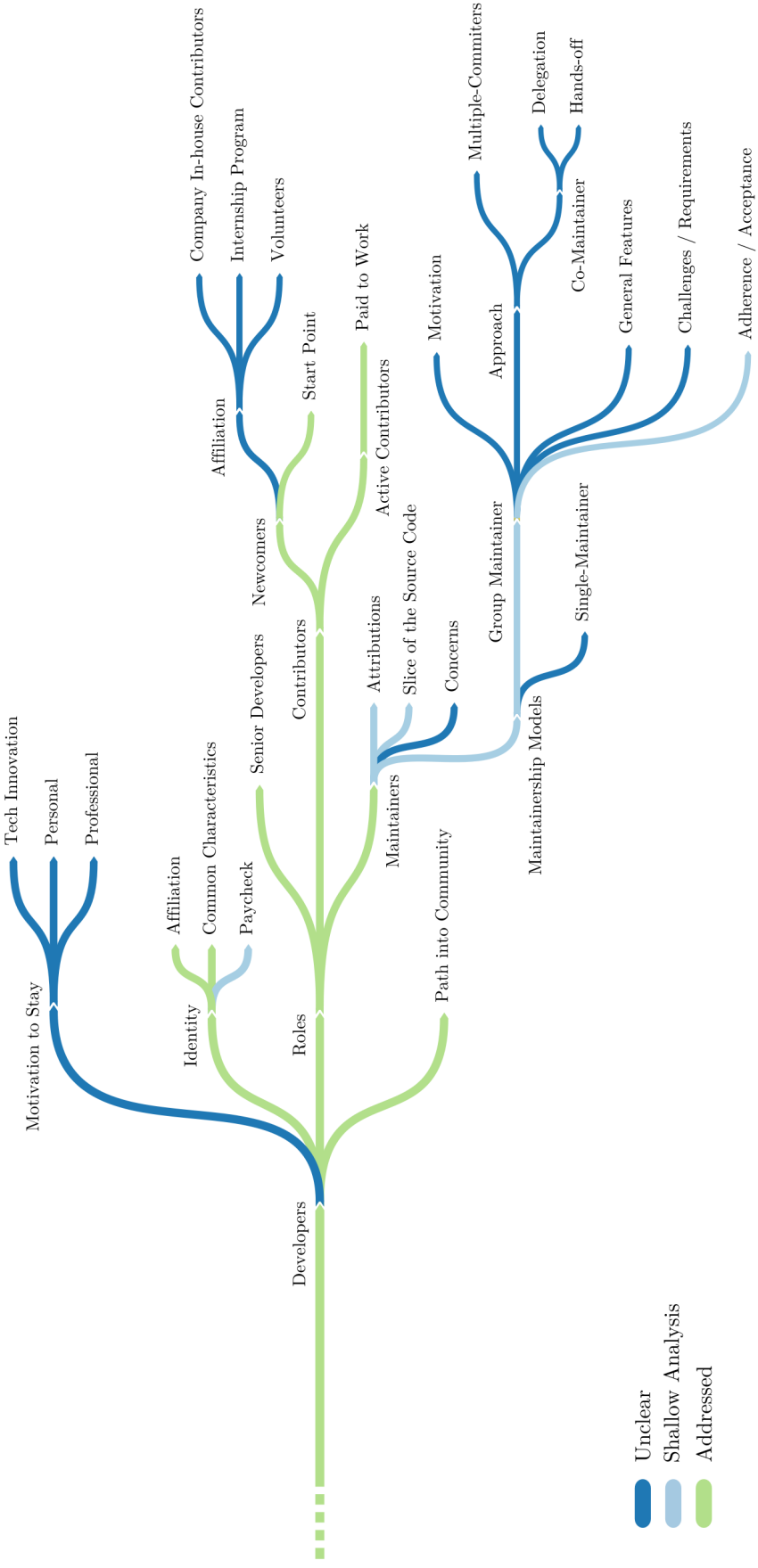


Figure 5.4: *Comparing interests of academia and community publications – Attributes regarding the Community Ecosystem (2) – Developers*

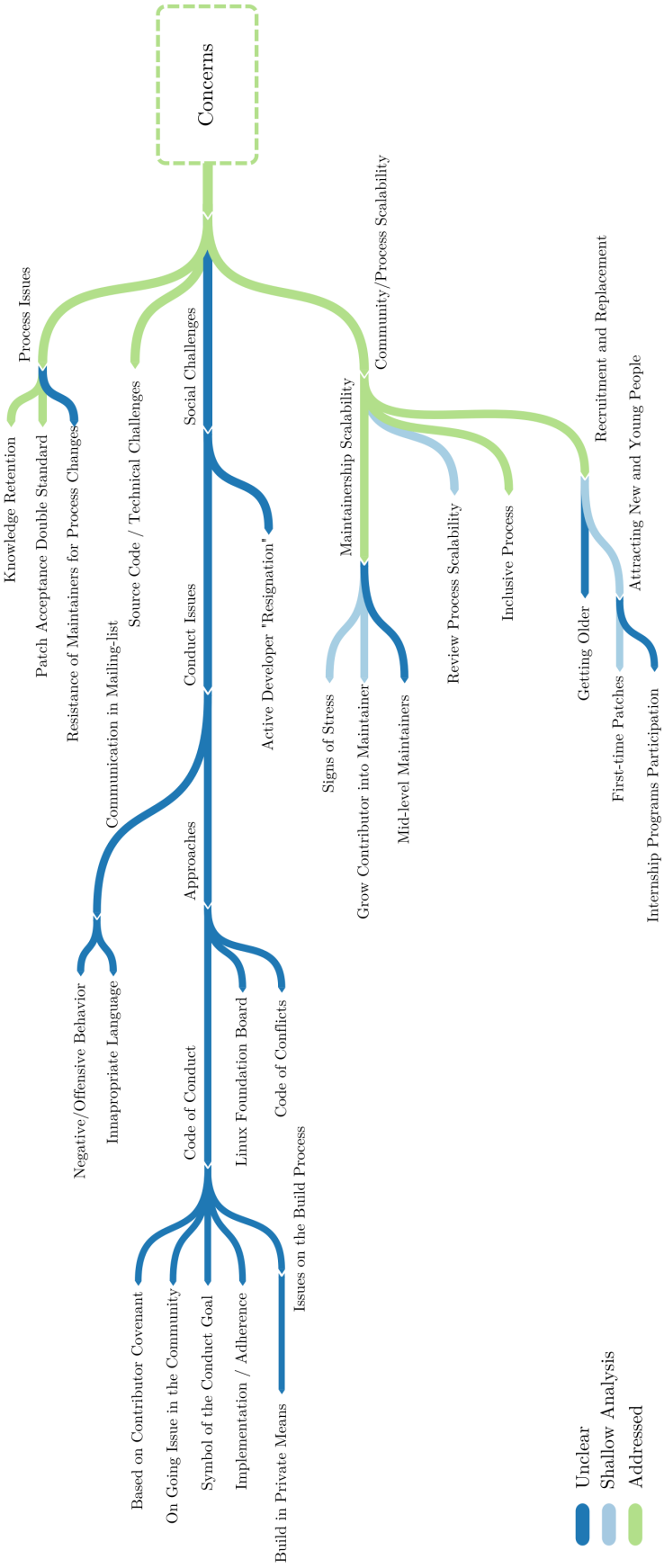


Figure 5.5: *Comparing interests of academia and community publications – Attributes regarding Community Concerns*

5.3 The Third Perspective - Participant-Observation

We conducted an Multivocal Literature Review considering community publications and software engineering studies on FLOSS to answer *RQ1. How do software engineering studies and Linux community publications describe the current Linux development community model?* The comparative mind map presented and discussed in Section 5.2 answers *RQ1.2. Do Software Engineering studies already cover these topics?* In this section, we enrich the discussion on MLR findings with data from a participant-observation that I conducted in communities of two Linux kernel subsystems, IIO and DRM. The triangulation aims to produce a solid discussion of our findings to more effectively answer *RQ3. What are the possible gaps and opportunities for academic research in FLOSS development topics?*

During the content analysis of grey literature, we decided to restrict the study scope to focus on the Linux Kernel Development Community. We made this decision because we could evaluate a good saturation and systematization of the concepts related to the Linux kernel development processes after the first round of GLR coding. Also, the Linux kernel community presents a good description of the main community's artifacts produced and delivered during the development and release processes.

We ratify this assessment by analyzing content from formal literature. On the one hand, we observed that software engineering studies reference some of the same grey documents to describe those elements. A good example is the text fragment from Izquierdo-Cortazar et al. (2017), *"All this process, since when a patch series is posted for first time to the mailing list, until when it is accepted to be committed is what we will refer to as the "code review process" for that specific patch series. The full process is described for developers in the Linux kernel documentation, section 'Submitting Patches'."* On the other hand, most of the selected studies converge on describing development processes, such as the Linux kernel patch flow, code review process, and the release process (mainline and stable versions).

The Linux kernel project has inspired Raymond (1999) to define the Bazaar model of software development management. However, both selected community publications and primary studies dissociate the current project from those characteristics described more than two decades ago as open-source project management standards. *"Importantly, Linux's development defies common management theory [..]. At the same time, it does not fit into any common software lifecycle model"* (Feitelson, 2012). *"The Linux community is a pyramid"* (T. L. Foundation, 2010) and the Linux kernel project is governed by dictatorship, and patches are passed up a "chain-of-trust" to the dictator (Rigby et al., 2014). The more the community grows, the more hierarchical it becomes. *"A number of subsystems are growing to the point where there needs to be some overall higher-level coordination. So there are more two and three-level trees than there used to be"* (Corbet, 2017a). In short, current studies and community publications do not resort to attributes regarding anarchy or flatness to describe the state of the Linux kernel development model.

My participant-observation in the Linux kernel community brought a third view on the findings from literature. The benefits of including this qualitative data collection method are twofold: expanding knowledge on characteristics of the community under investigation and capturing nuances and unspoken rules on Linux kernel development.

We assumed that some social aspects that shape the Linux kernel development model are undocumented or appears only eventually. Therefore, the field research could reveal these elements present in the daily tasks of development, maintainership and mentorship of active members.

The knowledge gained from this community engagement work is further detailed in Appendix C and Appendix D. The community immersive work and data collection process lasted 15 months (not sequential). However, once the investigation has completed, I continue to be part of the community and participating in activities related to development and maintenance.

In participant observation, the first observer task is to become a member of the community under investigation. To join the development community around a Linux subsystem, I, as a participant-observer, developed specific technical and non-technical skills to contribute to the project. This learning process started with a training phase on developing device drivers and sending patches to the IIO subsystem. The approach adopted in this research for training activities comprised of:

1. *Development environment setup* for coding and communication:
   (a) choosing a more accessible e-mail client to send patches;
   (b) using a bouncer to stay connected on IRC community channels;
   (c) identifying the maintainer's repository to contribute to a specific Linux kernel subsystem;
   (d) using a virtual machine and be able to modify, compile, install and test customized kernel versions.
2. *Learning development and release process*: how to patch and send contributions through subsystem mailing-list; checking code style rules; understanding the contribution flow.
3. *Getting involved*: sending code clean-up changes, examining staging drivers' debts and sending improvements and bug fixes.

I resorted to tutorials and integrated the FLUSP (FLOSS at USP)[4]. FLUSP is an extension group of the University of São Paulo (USP) created to foster student participation in FLOSS projects.

To become part of the group of community developers and understand the contribution workflow (submission, revision, acceptance, and merge), I developed a total of eight patches to the selected subsystem community, Industrial I/O. A summary of these contributions is presented in Appendix C. In the Linux kernel project, code, comments, reviews and other discussions use the same route: emails in the mailing list. Therefore, each code contribution sent to the mailing list can be considered a core of interaction between developers. Linus subsystem mailing lists are open and publicly available on the Internet, all contributions and e-mails were collected as data sources to identify members, roles and how interactions

[4] https://flusp.ime.usp.br

happen in the Linux kernel development community. I also took notes and wrote blog posts describing each cycles of contribution, lessons learned, and reflections[5].

Continuing my immersive work, I moved to a subsystem with a larger community of active contributors, the kernel DRM/GPU subsystem (DRI/DRM). Even being in the same project, each subsystem requires from the developer a specific computer knowledge base in terms of software and hardware. Besides the transition between operating system components, I needed to understand other specifications and testing tools for the drivers in the GPU subsystem.

I could experience different community roles in this phase. Each role brings a specific perspective of the development process and different levels of interaction with other members. During one year, I walked a path in the community as:

- Linux/DRM development community newcomer;
- Google Summer of Code intern;
- Virtual KMS driver maintainer;
- Outreachy internship program mentor

I sent 15 patches with different complexity of code contributions: code style, bug fixes, maintainability improvements, test coverage, operational behavior fixes, and new features. I also reviewed ten patches for DRM subsystem drivers. Finally, I posted 12 blog posts to describe my participation in the subsystem community. In addition to this involvement in more technical activities, maintainership and mentorship required more social skills, such as a careful speech, building a follow-up routine, understanding the other side perspectives, attention to conduct issues. The participant observation in DRM/Linux community is detailed in Appendix D. I also present discussions about the community joining phase, the difficulties experienced, and some reflections in the face of particular situations.

5.3.1 Mapping gaps in academic and community publications

Interestingly, grey and formal literature converge many times in carrying unclear information about Linux kernel software. In both samples of selected documents, we could find authors describing the software as an Operating System (OS) instead of an OS kernel. In other words, *"Linux is the kernel: the program in the system that allocates the machine's resources to the other programs that you run. [..] The ambiguous use of the name doesn't help people understand."* [6]

Some selected studies define the Linux project modularization based on directories and files, while community publications prefer to use the subsystems organization. In academic studies, statistics are grouped by year while community publications present data by kernel release. For Avelino et al. (2018), investigation at the subsystem level *"requires a reference architecture of the Linux kernel, as well as a mapping between elements at the source code level to elements in the architectural model."* Therefore, out of the typical approach in

[5] https://melissawen.github.io/blog/2019/04/10/first-patch-linux
[6] https://www.gnu.org/gnu/linux-and-gnu.en.html

directories and years, they *"use the mapping rules set by Greg Kroah-Hartman, one of the main Linux kernel developers"* and available in his repository as scripts[7] to map files in each subsystem and different kernel releases. The logs and scripts in his repository is explicitly the same used by the Linux Kernel Development Report 2012 by Linux Foundation.

We also find divergences in study findings and community claims. While maintainers have frequently advertised through community publications signs of stress, the selected study on maintainership scalability, Zhou et al. (2017) bring another perspective about this concern. They investigated the kernel maintainer activities and *"found that the workloads of the average maintainer and the median maintainer do not appear to increase, thus some risks hypothesized in the community are not evident"*. Our participant-observation brings a third point of view. The study has not considered that active developers and mainly maintainers are usually employees from companies that do not pay them to work exclusively for the kernel. Moreover, many of them have few opportunities to upstream contributions under their employer's support. Consequently, maintainership could be an extra work that contributes to boosting the career and also increasing their workload.

In their work, Zhou et al. (2017) mentioned a community publication that we have also analyzed in our GLR (Corbet, 2016e): *"Linus Torvalds said that he has come to like the group maintainer model, where more than one person takes responsibility for a given subsystem. However, numerous developers were skeptical of the idea"*. We have analyzed this document and its snowballing references, and, from our understanding, the developers skepticism is around the sufficiency of a model with two maintainers. In addition, Torvalds has refused the idea of sharing commit-rights in the top-level maintainance.

After a participant observation in the Linux kernel project, we found statements in the primary studies' that do not fit with practice. After more than a year of involvement with the DRM subsystem community performing distinct roles (volunteer, intern, maintainer, and mentor), I observed the developer preference on sending patches that change only parts of the subsystem only to specialized mailing-lists, and do not include the project-wide mailing list linux-kernel@vger.kernel.org as recipient, see examples at Appendix D. This practice diverges from the assumption of Tan and Zhou (2019) to avoid obtaining duplicate patches for data collection, considering that *"In general, LKML contains all the patches because the kernel community stipulates that the patches sent to those specialized lists need to be copied to LKML."*. Also, the Linux kernel Documentation[8] states that *"linux-kernel@vger.kernel.org functions as a list of last resort, but the volume on that list has caused a number of developers to tune it out."*. In summary, *"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"* (Bettenburg et al., 2015).

To overcome the data collection challenge in defining a unique developer identity, Avelino et al. (2018) *"first assign to a single developer all commits with the same e-mail, but with different names"*. However, we found examples in the subsystems community that show that alias approach is not enough. The same person has more than one e-mail address, commonly distinguishing personal and professional relationships by e-mail

[7] https://github.com/gregkh/kernel-history

[8] https://www.kernel.org/doc/html/latest/process/submitting-patches.html

domain. Moreover, the same person usually switches between personal and professional e-mail addresses to grant the contribution copyright to an outside organization that sponsors the work.

Avelino et al. (2018) claim that file authorship measures help identify each file's key developers. This claim seems valid for developers who are still active in the community. Nevertheless, some developers and companies do not continue contributing, and the code ends up being maintained by other developers. An example is the VKMS driver. I am currently co-maintaining it, and one of the driver's authors decided not to continue working on the community and have sent contributions for more than two years. From the field research experience, identifying the file maintainers is more accurate than authors. Another approach would be checking the number of lines by last change authors using git commands, like git blame.

Capiluppi and Izquierdo-Cortázar (2013) verified commit date and timestamp on git to assess developers activity within the Linux Kernel during the day and the week. They claim to carefully monitor activities on late night and after office time slots since they are more likely to introduce additional complexity to the project code. In this scenario, two issues deserve further investigation: (1) communication means should be considered as a data source, and (2) code contribution effort in high-complex changes is a long-term production. Therefore, commit timestamp is a small piece of the work effort and would be inaccurate data to determine when development happens.

Although the community claim that *"developers who work on independent projects tend to think of themselves as affiliated with the project first, and their employer second; that results in a strong incentive to avoid compromising the project's goals in favor of what today's employer wants"* (Corbet, 2016a), this neutrality or care to the Linux kernel project is not clear. Some participants expressed the desire to remain in the community when they leave their jobs but continuing as a volunteer is not as frequent as it seems. On the other side, others have expressed they are working in the Linux kernel project because it pays well, and the skill brings professional recognition to high job positions.

Tian et al. (2012) warned of the challenges and cautions when using keywords for collecting and mining patch data. For them, *"All of these studies employ a keyword-based approach to infer commits that correspond to bug fixes, typically relying on the occurrence of keywords such as 'bug' or 'fix' in the commit log. Some studies also try to link software repositories with a Bugzilla by the detection of a Bugzilla number in the commit log. Unfortunately these approaches are not sufficient for our setting because: 1) Not all bug fixing commit messages include the words 'bug' or 'fix' [..]"*. Eyolfson et al. (2014) also point out the misuse of 'fix' keywords in the Linux and other two FLOSS projects investigated, and how false positives affects precision and recall of their methods. During participant observation, we observed the misuse of these keywords in different situations: (1) in the training and insertion stage in the IIO community, as a member of FLUSP; (2) in the internship for the DRM community, as a patch reviewer. What has been observed is that new contributors use the word fix to describe clean-up codes, legibility improvements, or debugging messages. For bug-fix patches, the Linux kernel Documentation recommends that *"if your patch fixes a bug in a specific commit, e.g., you found an issue using git bisect, please use the 'Fixes:' tag"*. The tag helps determine the bug's origin, review a bug fix, and assist the stable kernel

team. Unfortunately, it is a recommendation that still depends on human attention to ensure its employment.

Ultimately, the participant-observation in the Linux kernel development community enables us to reinforce and discuss our findings with the qualitative perspective. In the context of large-scale FLOSS projects, we believe that becoming a member of the studied community brings the researcher a better understanding of undocumented rules and roles that shape development processes. Uncovering these elements is relevant to conduct and increase the quality of the research design and discussion of results. Despite the reported challenges for GLR, we also argue that the use of GL material is even more fruitful in FLOSS research, given the vast amount of public information resources produced by the FLOSS communities. In such a scenario, we claim that researchers should investigate such resources before taking the time of FLOSS contributors with surveys or interviews (V. Garousi et al., 2016), especially considering that *i)* most of the needed information is already available with rich details and *ii)* top contributors usually do not have much time for extensive interviews.

5.3.2 Threats to Validity

The Linux kernel software and its community-project have peculiarities compared to other FLOSS projects. Linux is both a typical case of a successful FLOSS project and an extreme case of vigor in the publications made by its community. Due to the software type, its complexity, longevity, and project community size, the Linux kernel is the topic of many publications. It has a high availability of materials produced by practitioners, unlike many other FLOSS projects. Therefore, obtaining information and performing some GLR steps can vary in complexity when applied to other FLOSS projects' investigations.

On the one hand, the maturity and variety of market usage of the Linux kernel project in the software industry increase the amount of information available public and online. On the other hand, the Linux development process's peculiarities and the community structure demands familiarity with FLOSS practices to interpret the data. To expand this knowledge and increase the accuracy in coding the data, a researcher should get closer to the community's daily life, searching for complementary documents and observing the community in practice.

This approach brings benefits but can lead to bias. Nevertheless, due to the Linux project relevance and protagonism in the last two decades of FLOSS research, an accurate mapping of its current state of development practices could improve the execution and relevance of software engineering research on FLOSS.

The Linux kernel development model was the case for three different methods of qualitative data collection: grey literature review, systematic literature review and participant-observation. Although we followed a systematic approach to searching and selecting documents for formal and gray literature, the differences in quality of the search tools and vocabulary used by different audiences led to specific adaptations in each review. Therefore, it was impossible to use the same document search and selection protocol to review both kinds of literature.

We used the same time range and search-strings as close as efficient to select 108

documents from grey literature and 25 software engineering studies that approach the Linux kernel development community's characteristics. Despite the difference in number, most of the grey materials are brief texts, while the selected studies have at least ten well-structured pages. After we have analyzed both content, we considered to reach balanced sources of data, notwithstanding the large difference in number would affect the diversity information equilibrium.

We resorted to a multivocal literature review to highlight convergence and divergence of interest by the state-of-the-art and state-of-the-practice in ten years of Linux kernel development. We considered ten years a reasonable time range of review. Our initial sample began from thousands to 340 pre-selected grey documents, and 1081 search hits in academic digital libraries. Nevertheless, we believe that every open topic deserves more in-depth and more centralized literature examination to assess Software Engineering studies' coverage more accurately.

Finally, we use the qualitative method of participant-observation, aware of the researcher's risk of bias. We also weighted the power of the inclusion of a not usual perspective on Software Engineering studies on FLOSS. Therefore, we considered that the benefits of understanding the concepts and expanding the critical view on the study topics are superior to the risk of inserting a bias intrinsic to a human researcher.

Chapter 6

Conclusion

Software engineering literature still presents little of what is performed by FLOSS practitioners, their activities, and how often they happen. This distancing from the practice leads academic works to lack a well-understanding of the FLOSS phenomenon, carrying a homogeneous and biased perspective. From our investigation results, we claim that software engineering research must accept and analyze knowledge artifacts produced by practitioners to enrich knowledge about FLOSS development. It enables to capture a more up-to-date snapshot of the project development and challenges currently faced in the daily activities. Also, grey literature brings the nuances of the real world from the practitioners' point of view.

Alongside commercial and academic publications, GL is used to broaden the academic understanding of how FLOSS developers interpret their daily work environment. Despite the informality of GL, it brings advantages of its own. Studies with negative or null results are easier to find in GL than in peer-reviewed academic literature, enabling a more critical perspective, with a potential reduction of bias and visualization of more balanced evidence (Paez, 2017). GL is also a leading source for identifying topics and gaps not yet covered by academic literature. It also enables the investigation of more up-to-date and emerging information since academic studies incur long publication delays due to the peer-review process.

Bearing this in mind, we reviewed the grey literature and systematized the Linux kernel community characteristics in a mind map where concepts are grouped in three main topics: General Characteristics, Ecosystem, and Concerns. With this mapping, we answer ***RQ1. How do software engineering studies and Linux community publications describe the current Linux development community model?*** and *RQ1.1. What attributes practitioners use to characterize the Linux development community? What are the current social and organizational challenges from the community perspective?*. Consequently, the resulted mind map provides **C1. A comprehensive characterization of the contemporary Linux kernel development community.** and explains **C2. Community characteristics and social-technical nuances that shapes a FLOSS project development**. It may also help FLOSS researchers evaluate if the Linux project fits the characteristics desired for their investigation.

We combined several data collection methods to provide a comparison of our findings from academic and community publications that answer *RQ1.2. Do Software Engineering studies already cover these topics?*. With a comparative mind map and discussion from development trenches, we elucidated the pace between the Linux kernel community and Software Engineering studies and deliver **C4.1 A map of potentially misleading information gaps between theory and practice in the contemporary Linux kernel development community.**

Because GLR is still a nascent trend, just a few software engineering reviews have already included GL materials (Bailey et al., 2007; França et al., 2016; Soldani et al., 2018). Although there are recommendations for GL material inclusion in literature reviews, there is not yet a precise methodology or guidelines prescribing well-defined steps and restrictions for conducting Grey Literature Reviews (GLR). Some researchers have mitigated this problem by applying systematic review methods for examining GL. Nevertheless, conducting GLR still requires adaptations for handling the variety in quality, size, types, and structures of documents, besides the larger volume of available materials compared to a traditional literature review. Moreover, the reviewer must deal with the lack of robust search engines and textual and natural language analysis challenges.

Hence, we reported in Chapter 4 a set of mechanisms for data collection and analysis to systematically examine grey materials produced by members of a FLOSS ecosystem. This investigative design answers our ***RQ2. What research techniques can be used to examine a FLOSS project through its community publications?*** and contributed with **C3. Guidelines to examine FLOSS projects through its community publications.** We overlapped data from GLR, SLR, and Participant-Observation to identify topics with misunderstandings or shallow coverage on FLOSS studies that deserve more in-depth investigations. This multi-method approach presents **C4. A combination of research strategies that could boost research on FLOSS ecosystems** and answers ***RQ3. What are the possible gaps and opportunities for academic research in FLOSS development topics?***

6.1 Future Work

Our work shows that grey Literature is a leading source for identifying topics and issues from the real world not yet covered by academic literature. We believe that the use of grey documents enables a researcher to start from a more prosperous basis by capturing parts of the practice. However, the lack of data contrast is dangerous due to the power of incorporating more bias and misleading information for research design and, consequently, skewed results since personal opinions are intricate to this information source.

From our content analysis and critical discussion, FLOSS researchers may find opportunities to conduct in-depth investigations of some Linux kernel practices and expand state-of-art and -practice even more. Future works should address an explicit comparison of each of the concepts obtained from multivocal literature review to the third perspective of participant observation in the Linux kernel development community.

The selected grey documents, practitioners' also revealed particular concerns about most code change authorship in the Linux kernel project being concentrated in top contrib-

utors. These contributors are usually subsystem maintainers and also employees of large companies. In this scenario, continuing to examine the influence of outside organization in the project is essential, *"not to malign the sponsorship or participation of corporations or governments in FOSS, but to show how these may skew the dynamics of a FOSS project"* (Forrest et al., 2012).

We observed issues in the current Linux kernel development model that are little discussed in formal literature. First, the current code maintenance organization and how it impacts active developers. This subject includes issues in the maintainer's routines, such as signs of stress, burn-out, fears, paralysis in the face of innovations, and topics regarding the hierarchy of commit-rights, the double-standard in patch reviews and commits, and the distinct models of maintainership per subsystem. Future FLOSS research should address an in-depth investigation of these concepts to identify the challenges and benefits of changing the maintainership model (between the single, group, and multiple-committers) to support processes scalability and mitigate bottlenecks suggested by traditional and grey literature in the current pre-commit review process.

Second, we mapped research opportunities regarding community subsistence: the difficulty in transferring knowledge and the steep learning curve. Both contribute to the aging of the community and missing younger voluntary contributors. On the one hand, these issues may be addressed by companies developing new contributors in-house. On the other hand, abusive culture on communication, conduct issues, and a toxic environment would affect new and active developers. In addition, future works should consider other means of communication and decision-making besides the mailing-lists, such as IRC channels and conferences.

Lastly, FLOSS research should benefit from investigations of the source code's continuous growth and the impacts on source code and processes of embracing the non-regression rule. Also, studies should evaluate the significance of the code change authorship centralized in top contributors, usually maintainers and large companies' employees. All these issues are mapped in this thesis and deserve further scrutiny by the research community.

Our claimed contributions and the wealth of information in this work reveal the importance of using investigative methods able to embrace the continuous transformations of the FLOSS phenomenon and the social aspects of community-based development. The review of community publications and direct participation in developing a project as a community member are strategies not commonly used in the studies of Software Engineering in FLOSS. In our work, the use of Multivocal Literature Review and Participant-Observation brought academia closer to the everyday practice of developing the Linux kernel, collecting richer and more up-to-date data, and identifying undocumented social aspects and unspoken rules. Given the results obtained here, we understand that these methods fit very well in the characteristics of the development of FLOSS and should be strongly considered in further research in the area.

Appendix A

Grey Literature Review Protocol

We considered Grey Literature any online publication not peer reviewed. The term ‘grey literature’ is often used to refer to reports published outside of traditional commercial publishing. (Cochrane)

Motivation

Some studies of software engineering have presented a skewed view of the FLOSS phenomenon. Among the possible biases, we highlight the generalization of FLOSS projects (black-and-white view), distance from day-to-day practice, and the belief that FLOSS projects do not have well-defined management structure and workflow methods. Although there is a wealth of artifacts produced by communities to document the daily practice of FLOSS and outsource discussions and reflections, there is no academic study focused on summarizing the concepts disseminated by FLOSS communities about their development models. This review intends to fill this gap by understanding how FLOSS practitioners characterize the development model of the community that they contribute.

A.1 Research Questions

RQ1. How do Linux community publications describe the current Linux development community model?

RQ1.1. What attributes practitioners use to characterize the Linux development community? What are the current social and organizational challenges from the community perspective?

RQ2. What research techniques can be used to examine a FLOSS project through its community publications?

A.2 Initial steps

1. Searching for publications about grey literature review in software engineering

2. Brainstorm sources and potential database to search grey literature on FLOSS

A.2.1 Data Sources

Textual	Audio-visual
• Foundations and Project Webpages • Industry-oriented magazines • Books • Experienced Developers Blogs • Whitepapers • Mailing-list	• Movies • Videos on Youtube • Podcasts • TED Talks

Table A.1: *Types of Grey Literature Datasource*

A.2.2 Identifying the Database

Textual

Targeted websites (recommendations from FLOSS experts; Locating organization via Google)

- https://opensource.org/
- https://www.fsf.org/
- https://www.kernel.org
- https://www.linux.com
- LWN.net
- https://kernelnewbies.org/
- https://www.linuxfoundation.org/
- https://www.linuxinsider.com
- http://planet.kernel.org/

Industry-oriented Magazines

- https://www.wired.com/
- http://www.drdobbs.com/
- https://www.linuxformat.com/
- http://www.linux-magazine.com/
- https://www.linuxjournal.com/
- https://opensourceforu.com/
- https://www.networkworld.com

Books

- Free as in Freedom
- Producing OSS
- Understanding the Open Source Development Model

Experienced Developers Blogs

- http://www.linux-magazine.com/Online/Blogs
- https://blog.ffwll.ch/
- http://www.kroah.com/log/
- https://sage.thesharps.us/

Whitepapers

- http://www.findwhitepapers.com/technology
- https://www.bitpipe.com/

Google advanced search

GL databases

- http://www.opengrey.eu
- https://arxiv.org/

Consultating with experts

Snowballing references and backlinks

Audio-visual

- https://google.com
- Movies: Revolution OS, The Code (Documentary), Pirates of Silicon Valley, The 5 Keys to Mastery
- https://www.youtube.com
- Podcasts: papolivre.org; Sunday Morning Linux Review (http://smlr.us/); Free as in Freedom (http://faif.us/); https://twit.tv/shows/floss-weekly; KernelPodcast(http://www.kernelpodcast.org/);
- https://www.ted.com

A.3 Planning GLR

Relaxed Search String

Inclusion and exclusion criteria

Interval time: 2009-2019+

Table A.2

Linux	Development	Model
Linux Kernel	Business	Structure
Linux Project	()	Scheme
		Rules
		Process
		Guide
		Community
		Culture
		()

Table A.3

Inclusion criteria	Exclusion criteria
Published online by practitioners or researchers in the FLOSS area	Published by enthusiasts who have not participated in any FLOSS-related projects or any FLOSS research
Available in English	Unavailable in English
Most current version of the document	There is a new (updated) version of the document
Describes the development of Linux kernel: reporting practices and/or presenting statistics and/or discussing management and rules inside the project and/or studying the project development and/or its community	Did not contain any reference of development practices of Linux project
	Merely discusses technical issues / coding / new features

Ranking Resource-Types

A.4 Search process

A.4.1 Database Selection

Searching Data Source

Search on Google for relevant websites and contact experts to create a list of targets

Classifying Data Source

A.4.2 Searching documents

1. When a search tool is available on the selected database, apply the search string
2. Otherwise, search by hand relevant contents
3. Export results to an spreadsheet
4. Remove duplication
5. Highlight title appeared relevant and analyze abstract, when available
6. Author profile: experience, role in FLOSS project, active or not
7. Publisher history/profile
8. Snowballing references

A.5 Selection process

We based on guidelines provided by Higgins and Green (2008).

1. Use (at least) two people working independently to determine whether each study meets the eligibility criteria.
2. Whener possible, screening of titles and abstracts to remove irrelevant reports (should be done in duplicate by two people working independently but it is acceptable that this initial screening is undertaken by only one person).
3. Two people working independently are used to make a final determination as to whether each study considered possibly eligible after title / abstract screening meets the eligibility criteria based on the full text of the study report(s).
4. It is important that at least one author is knowledgeable in the area under review, it may be an advantage to have a second author who is not a content expert.
5. Disagreements about whether a study should be included can generally be resolved by discussion.
6. A single failed eligibility criterion is sufficient for a study to be excluded from a review. In practice, therefore, eligibility criteria for each study should be assessed in order of importance

7. Pilot test the eligibility criteria on a sample of reports
8. The selection process must be documented in sufficient detail to be able to complete a flow diagram and a table of 'Characteristics of excluded studies'

Processes of documents identification, screening and content analysis are detailed in Chapter 4

Appendix B

Multivocal Literature Review Protocol

Multivocal Literature Review considers traditional commercial and non-commercial publications to build a comprehensive understanding of a specific research field.

For this, we opted for the gray literature review as a starting point and subsequently performed a systematic literature review. In this systematic review of the literature, the search and selection protocol for publications followed conventional patterns, already well established by software engineering research. From the publication sample obtained by applying the protocol, we established a comparative content analysis of the findings of the grey literature review.

B.1 Systematic Literature Review Protocol

Research Questions

RQ1. How do software engineering studies describe the current Linux development community model?

RQ1.2. Do Software Engineering studies already cover these topics?

RQ3. What are the possible gaps and opportunities for academic research in FLOSS development topics?

Data Sources

- Journal (Name of journal; Years searched; Any issues not searched)
- Peer-reviewed Conference Proceedings (Title of proceedings; Name of conference (if different); Title translation (if necessary); Journal name (if published as part of a journal)

Database

- IEEExplore

- ACM Digital library:
- ScienceDirect/Elsevier (www.sciencedirect.com)
- Springer (https://link.springer.com/)
- Google scholar (scholar.google.com) [For double check]

Search string

After testing the search string presented in Table A.2, we verify that some words combinations were more effective. It resulted in a condensed search string, accepted by all database selected to collect software engineering research publications.

Full text: "Linux kernel" AND ("Development Model" OR "community Practices" OR "development Practices" OR "community Practices" OR "Development Process")

Eligibility criteria

Interval time: 2009-2019+

- Published online in the selected database
- Available in English
- Most current version of the document
- The Linux project is subject of study
- Describe elements of Linux kernel development: practices and/or artifacts and/or processes and/or community characteristics
- Include discussion on the Linux kernel community and members, not only technical issues: coding and/or new features

Search process

1. Test the efficiency of the search string by checking the search hits.
2. Define an efficient search string.
3. Apply the search string in the search tool on each database.
4. Export results to Mendeley.
5. Remove duplication in the same database and cross-database.
6. Export references to spreadsheets for filtering.
7. Evaluate the relevance of each venue.
8. Remove papers from venues with low relevance to the FLOSS and SE area.
9. Screen title. One reviewers.
10. Remove papers with title clearly out FLOSS/SE out of context the FLOSS and SE area.
11. Screen title and abstract. Two reviewers in parallel. Remove by exclusion criteria.

12. Full-text read to select primary studies.

Selection process

We based on guidelines provided by Higgins and Green (2008).

1. Use (at least) two people working independently to determine whether each study meets the eligibility criteria.
2. One person screening titles and abstracts to remove irrelevant reports. Cochrane states that this step should be done in duplicate by two people working independently but it is acceptable that this initial screening is undertaken by only one person.
3. Two people working independently are used to make a final determination as to whether each study considered possibly eligible after title / abstract screening meets the eligibility criteria based on the full text of the study report(s).
4. It is important that at least one author is knowledgeable in the area under review, it may be an advantage to have a second author who is not a content expert.
5. Disagreements about whether a study should be included can generally be resolved by discussion.
6. A single failed eligibility criterion is sufficient for a study to be excluded from a review. In practice, therefore, eligibility criteria for each study should be assessed in order of importance
7. Pilot test the eligibility criteria on a sample of reports.
8. The selection process must be documented in sufficient detail to be able to complete a flow diagram and a table of 'Characteristics of excluded studies'

B.2 Content Analysis - Multivocal Literature Review

B.2.1 Content Analysis

Quantitative analysis

1. Extract raw text content from every paper selected;
2. Using a script for Natural Language Processing, plot the Word Cloud resulted for comparison and acquaintance;
3. Also take the word frequency, showing top words;
4. Finally, using the mind map resulted from Grey Literature review and the script processing, pre-define an initial group of core concepts.

Qualitative analysis

We based on a directed approach of content analysis (Hsieh and Shannon, 2005) for coding.

1. Use the GLR mind map and SLR Word Cloud findings as guidance for initial codes.
2. For each publication selected, derive coding categories from the text data. This step includes counting and comparisons of GLR map keywords and statements of the examining document, in addition to interpretate the subjacent context.
3. Verify the application (or not) of the concepts identified in the publication in each GLR mind map branch.

Appendix C

Getting Involved in the Linux Kernel Community

A distributed,large community of users and developers surrounds the Linux kernel project. The community is a place where users and developers can report problems, obtain code review and testing and expert advice. Besides, it is the gateway for changes to the kernel code (Love, 2010).

To join any community, a new member needs to assimilate the workflow and some basic rules of interaction. These information is available in the Linux kernel Documentation (https://www.kernel.org/doc/html/latest/process/howto.html), and in books, such as, the Linux Kernel Development of Love (2010). Besides, a new member should know the community norms to preserve a welcome environment. Members and new members must respect each other and avoid negative behavior in the community, as mentioned in https://www.kernel.org/doc/html/latest/process/code-of-conduct.html. Moreover, due to the software's singularities, a newcomer should develop skills to correctly change a piece of the system code.

C.1 Training Activities - Development Environment Setup

C.1.1 Basic Setup - This Research Approach

Setup e-mail client: This step consists of installing and configuring an e-mail client that allows the use of the terminal to send contributions managed by git version control system. The researcher selected Neomutt[1] as the client.

Join the IRC channel of the selected project: Here is the place for quick discussions and questions/answers with other active developers. For the IIO subsystem, developers meeting in #linux-iio on OFTC network.

[1] https://neomutt.org/

Clone the kernel git repository: Because each subsystem maintainer has a commit tree to accept works related to a particular subsystem, the contributor must keep the selected subsystem maintainer tree up to date. The git trees to contribute have a different location, and they are specified in the MAINTAINERS file: https://www.kernel.org/doc/linux/MAINTAINERS

Install a virtual machine for development: it prevents that, by some carelessness, the contributor damages its operating system. This care is necessary because by modifying and compiling a Linux image, the developer can replace the image of his/her system and insert a critical bug.

Learning how to modify, compile, and install a Linux kernel image: consists of a series of commands and settings that need to be executed or updated to perform the proposed work. I used the Kworkflow tool, a set of scripts to automate commands executed in the daily development activities: https://github.com/kworkflow

C.1.2 Understanding the process of sending contribution

Learn the Linux kernel development process: The Linux kernel development process is well-described in the project documentation at https://www.kernel.org/doc/html/latest/process/2.Process.html

Learn the code style rules: The project has code standardization rules described at https://www.kernel.org/doc/html/latest/process/coding-style.html. The community also developed tools to verify compliance with these rules.

Execute tutorials to introduce newcomers to the contribution flow: The Kernel Newbies website provides tutorials to foster newcomers' participation in the Linux kernel community. These tutorials could be found at https://kernelnewbies.org/FirstKernelPatch

C.2 Getting involved in the IIO subsystem

Code clean-up: Find the file of a drive. Run code check tools like checkpatch.pl or Coccinelle. These tools reveal easy-level issues and guide a newcomer to correct the code follow the code style rules.

Exploring staging drivers: Drivers in the staging tree are in the main Linux kernel source tree, but, for technical reasons, it was still not merged into the main portion of the Linux kernel tree[2] Therefore, developers can improve these drivers beyond code style. To develop substantial improvements, a developer should search for the driver documentation. An example is a Datasheet of the IIO-driver AD7150 from Analog Devices: https://www.analog.com/media/en/technical-documentation/data-sheets/AD7150.pdf

Contributions sent to the Industrial I / O subsystem

[2]https://lwn.net/Articles/324279/

1st Version Date	Commit message	Developer(s)	Kind of change	# Versions	Status
Apr 2, 2019	staging: iio: frequency: ad9834: Remove unnecessary parentheses	Melissa Wen	Code style	1	Merged
May 3, 2019	staging: iio: ad7150: organize registers definition	Melissa Wen	Improve readability	1	Declined
May 3, 2019	staging: iio: ad7150: use FIELD_GET and GENMASK	Melissa Wen	Improve maintanability	2	Merged
May 3, 2019	staging: iio: ad7150: simplify i2c SMBus return treatment	Melissa Wen	Reduce verbosity	2	Merged
May 3, 2019	staging: iio: ad7150: clean up of comments	Melissa Wen	Improve readability	2	Merged
May 18, 2019	staging: iio: cdc: ad7150: create of_device_id array	Barbara Fernandes, Wilson Sales Melissa Wen	Create missing element	1	Merged
May 18, 2019	staging:iio:ad7150: fix threshold mode config bit	Melissa Wen	Bug fix	1	Merged
Jun 14, 2019	staging: iio: ad7150: use ternary operating to ensure 0/1 value	Melissa Wen	Remove idiom, Make operation consistent	1	Merged

Table C.1: *List of contributions sent to IIO Linux subsystem*

C.3 Anatomy and contribution flow: the case of [PATCH] staging:iio:ad7150: fix threshold mode config bit

Proposing a code change involves a series of interactions. Each patch submission improves the acquaintance with processes, standards established, communication style among members, and roles played by each in code review and merging process. The Linux kernel documentation already presents a Howto for sending a code contribution[3].

C.3.1 Sending a contribution by e-mail

In this section, we present a real-submission patch e-mail and point out recommendations from this research experience.

Header Information

In addition to submitting a contribution to the appropriate mailing list for the related file, the recipient list must include the driver maintainers and others who may be interested in reviewing the proposed changes.

Date: 18 de maio de 2019 22:04
From: M...
To: L...
Cc: linux-iio@vger.kernel.org, devel@driverdev.osuosl.org, linux-kernel@vger.kernel.org

Change Description

[3] https://www.kernel.org/doc/html/latest/process/submitting-patches.html

The commit message should also describe what is modified and the reasons for this change and reference documents or tools that support the proposed change. In this case, the developer cited the driver configuration description presented in the driver datasheet.

"According to the AD7150 configuration register description, bit 7 assumes value 1 when the threshold mode is fixed and 0 when it is adaptive, however, the operation that identifies this mode was considering the opposite values.

This patch renames the boolean variable to describe it correctly and properly replaces it in the places where it is used."

Special Tags

For some cases, the developer must include tags already standardized. The list of tags is presented at https://www.kernel.org/doc/html/latest/process/submitting-patches.html#when-to-use-acked-by-cc-and-co-developed-by and https://www.kernel.org/doc/html/latest/process/submitting-patches.html#using-reported-by-tested-by-reviewed-by-suggested-by-and-fi Tags give the credits for contributors involved in the code change, beyond the author of the patch.

Fixes: 531efd6aa0991 ("staging:iio:adc:ad7150: chan_spec conv + i2c_smbus commands + drop unused poweroff timeout control.")

Commit Author(s) Signed-off-by: Melissa Wen <..@..>

Commit Diff —

```
drivers/staging/iio/cdc/ad7150.c | 19 ++++++++++——–
1 file changed, 11 insertions(+), 8 deletions(-)

diff –git a/drivers/staging/iio/cdc/ad7150.c b/drivers/staging/iio/cdc/ad7150.c
index dd7fcab8e19e..e075244c602b 100644
— a/drivers/staging/iio/cdc/ad7150.c
+++ b/drivers/staging/iio/cdc/ad7150.c
@@ -5,6 +5,7 @@
* Copyright 2010-2011 Analog Devices Inc.
*/

+#include <linux/bitfield.h>
#include <linux/interrupt.h>
#include <linux/device.h>
#include <linux/kernel.h>
@@ -130,7 +131,7 @@ static int ad7150_read_event_config(struct iio_dev *indio_dev,
{
int ret;
u8 threshtype;
```

```
- bool adaptive;
+ bool thrfixed;
 struct ad7150_chip_info *chip = iio_priv(indio_dev);

 ret = i2c_smbus_read_byte_data(chip->client, AD7150_CFG);
@@ -138,21 +139,23 @@ static int ad7150_read_event_config(struct iio_dev *indio_dev,
 return ret;

 threshtype = (ret >> 5) & 0x03;
- adaptive = !!(ret & 0x80);
+
+ /*check if threshold mode is fixed or adaptive*/
+ thrfixed = FIELD_GET(AD7150_CFG_FIX, ret);

 switch (type) {
 case IIO_EV_TYPE_MAG_ADAPTIVE:
 if (dir == IIO_EV_DIR_RISING)
- return adaptive && (threshtype == 0x1);
- return adaptive && (threshtype == 0x0);
+ return !thrfixed && (threshtype == 0x1);
+ return !thrfixed && (threshtype == 0x0);
 case IIO_EV_TYPE_THRESH_ADAPTIVE:
 if (dir == IIO_EV_DIR_RISING)
- return adaptive && (threshtype == 0x3);
- return adaptive && (threshtype == 0x2);
+ return !thrfixed && (threshtype == 0x3);
+ return !thrfixed && (threshtype == 0x2);
 case IIO_EV_TYPE_THRESH:
 if (dir == IIO_EV_DIR_RISING)
- return !adaptive && (threshtype == 0x1);
- return !adaptive && (threshtype == 0x0);
+ return thrfixed && (threshtype == 0x1);
+ return thrfixed && (threshtype == 0x0);
 default:
 break;
 }
 –
 2.20.1
```

C.3.2 Receiving feedback on mailing-list

From maintainer

Header Information

Date: 19 de maio de 2019 07:29

From: J...
To: M...
Cc: ..., linux-iio@vger.kernel.org, devel@driverdev.osuosl.org, linux-kernel@vger.kernel.org, kernel-usp@googlegroups.com

Comments inline On Sat, 18 May 2019 22:04:56 -0300
Melissa Wen [...] wrote:
> According to the AD7150 configuration register description, bit 7 assumes
> value 1 when the threshold mode is fixed and 0 when it is adaptive,
> however, the operation that identifies this mode was considering the
> opposite values.
>
> This patch renames the boolean variable to describe it correctly and
> properly replaces it in the places where it is used.
>
> Fixes: 531efd6aa0991 ("staging:iio:adc:ad7150: chan_spec conv + i2c_smbus commands + drop unused poweroff timeout control.")
> Signed-off-by: Melissa Wen <..@..>

Looks good to me. Applied to the fixes-togreg branch of iio.git pushed out as
as testing-fixes for the autobuilders to see if they can find anything
we have missed.

Thanks,

Jonathan

From an employee of the related device driver company

Header Information Date: 20 de maio de 2019 06:59
From: A...
To: J...
Cc: ..., linux-iio@vger.kernel.org, devel@driverdev.osuosl.org, LKML <linux-kernel@vger.kernel.org>, kernel-usp@googlegroups.com, ...

Comments inline On Sun, May 19, 2019 at 8:38 PM [...] wrote:
>
> On Sat, 18 May 2019 22:04:56 -0300
> Melissa Wen <..@..> wrote:

...

> > threshtype = (ret >> 5) & 0x03;
> > - adaptive = !!(ret & 0x80);

> > +
> > + /*check if threshold mode is fixed or adaptive*/
> > + thrfixed = FIELD_GET(AD7150_CFG_FIX, ret);

nitpick: i would have kept the original variable name as "adaptive",
mostly for consistency.
"adaptive" is used in other places as well;

as i recall, the fix is just oneliner in this case:

- adaptive = !!(ret & 0x80);
+ adaptive = !(ret & 0x80);

C.3.3 Receiving notification of merge

This section present the e-mails received after the patch acceptance that suggest the patch path from the subsystem maintainer tree to other Linux kernel trees related to staging drivers.

From Linux maintree maintainer

Date: 17 de junho de 2019 17:32
From: gregkh@...
To: m...

This is a note to let you know that I've just added the patch titled

staging:iio:ad7150: fix threshold mode config bit

to my staging git tree which can be found at
git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging.git
in the staging-linus branch.

The patch will show up in the next release of the linux-next tree
(usually sometime within the next 24 hours during the week.)

The patch will hopefully also be merged in Linus's tree for the
next -rc kernel release.

If you have any questions about this process, please let me know.

From df4d737ee4d7205aaa6275158aeebff87fd14488 Mon Sep 17 00:00:00 2001
From: Melissa Wen <..@..>
Date: Sat, 18 May 2019 22:04:56 -0300

Subject: staging:iio:ad7150: fix threshold mode config bit

According to the AD7150 configuration register description, bit 7 assumes value 1 when the threshold mode is fixed and 0 when it is adaptive, however, the operation that identifies this mode was considering the opposite values.

This patch renames the boolean variable to describe it correctly and properly replaces it in the places where it is used.

Fixes: 531efd6aa0991 ("staging:iio:adc:ad7150: chan_spec conv + i2c_smbus commands + drop unused poweroff timeout control.")
Signed-off-by: Melissa Wen <..@..>
Signed-off-by: Jonathan Cameron <..@..>
—
drivers/staging/iio/cdc/ad7150.c | 19 ++++++++++——–
1 file changed, 11 insertions(+), 8 deletions(-)

From Linux version maintainer

From: S[..]
Para: linux-kernel@vger.kernel.org, stable@vger.kernel.org
Cc: M..
From: Melissa Wen <..@..>

[Upstream commit df4d737ee4d7205aaa6275158aeebff87fd14488]

According to the AD7150 configuration register description, bit 7 assumes value 1 when the threshold mode is fixed and 0 when it is adaptive, however, the operation that identifies this mode was considering the opposite values.

This patch renames the boolean variable to describe it correctly and properly replaces it in the places where it is used.

Fixes: 531efd6aa0991 ("staging:iio:adc:ad7150: chan_spec conv + i2c_smbus commands + drop unused poweroff timeout control.")
Signed-off-by: Melissa Wen <..@..>
Signed-off-by: Jonathan Cameron <..@..>
Signed-off-by: Sasha Levin <..@..>
—
drivers/staging/iio/cdc/ad7150.c | 19 ++++++++++——–
1 file changed, 11 insertions(+), 8 deletions(-)

Appendix D

Path into Linux kernel Community

D.1 Changing to Another Subsystem - Development Environment Setup

My first barrier to become a member of the DRI community was setting up the development environment[1] and find how to contribute considering my lack of understanding of the subsystem functions and the necessary abstractions of graphic concepts.

I started to get involved in the community by exploring the subsystems drives and sending small contributions[2]. As I had already contributed to improving code style and cleanup checkpatch warnings, I chose a bug-fix issue to work on. I tried to reproduce the reported bug and looked for community guidance, but I was not successful[3]. *In this first experience, I realized that neither maintainers nor active developers know everything on a subsystem.* Consequently, considering my initial limitations, I went back to work on code style issues to keep engaged and find some mentorship.

D.1.1 Role: Google Summer Of Code Applicant

In a subsequent step of participant-observation, I decided to apply to the X.Org Foundation project on Google Summer of Code Program[4]. The application step consists of proposing a project based on the list of ideas suggested by community mentors of X.Org Foundation. In this program, mentors are active community developers that volunteer to guide new developers to accomplish their project goals. Graphic developers usually need hardware display capability to develop features and improvements. For this reason, I opted to work on VKMS[5], a software-only model KMS driver developed to run and test DRM and X in a headless machine.

[1] https://melissawen.github.io/blog/2020/01/08/hello-drm
[2] https://melissawen.github.io/blog/2020/03/04/first-patch-drm
[3] https://www.spinics.net/lists/dri-devel/msg248102.html
[4] https://summerofcode.withgoogle.com/
[5] https://dri.freedesktop.org/docs/drm/gpu/vkms.html

To figure out how to fix a bug on VKMS, I used a test suite originally developed by Intel, the IGT GPU Tools[6]. By checking how to fix a test case failure, I obtained an overview of the test coverage. I mapped the results of each test case and realized that the driver was presenting an unexpected behavior that cross-cut testing[7]. Therefore, I design a GSoC project to improve VKMS using IGT GPU Tools.

In one of my attempts to fix a bug on the VKMS driver, my patch received a review criticizing the proposed solution[8]. The reviewer proposed a better code to solve the issue, but I considered the message was rude and discouraging[9]. Initially, I thought the person was a DRI community member. However, after one year in the community, I can say this person is completely out of the usual Linux DRI more active developer. Maybe this developer came from another subsystem because I copied the Linux kernel mailing list.

Contributions sent to DRM subsystem for GSoC application

1st Version Date	Commit message	Developer(s)	Kind of change	# Versions	Status
Feb. 26, 2020	drm/amd/display: dc_link: code clean up on enable_link_dp function	Melissa Wen	Code style	2	Merged
Feb. 26, 2020	drm/amd/display: dc_link: code clean up on detect_dp function	Melissa Wen	Code style	2	Merged
Mar. 2, 2020	drm/amd/display: dcn20: remove an unused function	Melissa Wen	Improve maintainability	1	Merged
Mar. 21, 2020	drm/vkms: use bitfield op to get xrgb on compute crc	Melissa Wen	Bug-fix	1	Declined
March 21, 2020	drm/vkms: enable cursor by default	Melissa Wen	Improve usability	2	Merged
March 31, 2020	drm/amd/display: cleanup codestyle type BLOCK_COMMENT_STYLE on dc_link	Melissa Wen	Code style	2	Merged
March 31, 2020	drm/amd/display: codestyle cleanup on dc_link file until detect_dp func	Melissa Wen	Code style	2	Merged
March 31, 2020	drm/amd/display: code cleanup on dc_link from is_same_edid to get_ddc_line	Melissa Wen	Code style	2	Merged
March 31, 2020	drm/amd/display: code cleanup of dc_link file on func dc_link_construct	Melissa Wen	Code style	2	Merged

Table D.1: *Contributions sent to DRM subsystem for GSoC application*

D.1.2 Role: GSoC Intern

My project was the unique X.Org Foundation project accepted[10] in 2020 GSoC. This participation was hands-on, with technical issues and community engagement experience. I reported the steps in the learning curve and project acquaintance in 12 blog posts:

[6] https://gitlab.freedesktop.org/drm/igt-gpu-tools
[7] https://melissawen.github.io/blog/2020/03/23/a-tangle-issues
[8] https://www.spinics.net/lists/kernel/msg3450967.html
[9] https://melissawen.github.io/blog/2020/03/23/good-bad
[10] https://summerofcode.withgoogle.com/archive/2020/projects/6046648674811904/

Date	Title	Link
2020/05/13	I'm in - GSoC 2020 - X.Org Foundation	https://melissawen.github.io/blog/2020/05/13/im-in-gsoc
2020/05/20	Everyone makes a script	https://melissawen.github.io/blog/2020/05/20/community-bounding
2020/06/02	Status update - Tie up loose ends before starting	https://melissawen.github.io/blog/2020/06/02/status-update
2020/06/03	Walking in the KMS CURSOR CRC test	https://melissawen.github.io/blog/2020/06/03/overview_kms_cursor_crc
2020/06/15	Status update - connected errors	https://melissawen.github.io/blog/2020/06/15/status-update
2020/07/06	GSoC First Phase - Achievements	https://melissawen.github.io/blog/2020/07/06/first-round
2020/07/17	Increasing test coverage in VKMS - max square cursor size	https://melissawen.github.io/blog/2020/07/17/add-max-cursor-size.html
2020/08/12	The end of an endless debugging of an endless wait	https://melissawen.github.io/blog/2020/08/12/end_of_endless.html
2020/08/19	If a warning remains, the job is not finished.	https://melissawen.github.io/blog/2020/08/19/let-vkms-blend-it.html
2020/08/27	Another day, another mistery	https://melissawen.github.io/blog/2020/08/27/writeback-is-back
2020/08/28	Better validation of alpha-blending	https://melissawen.github.io/blog/2020/08/28/translucent-cursor-testcase
2020/08/31	GSoC Final Report	https://melissawen.github.io/blog/2020/08/31/gsoc-final-report

Table D.2: *blog_posts*

The experience of blogging and sharing the content in the Freedesktop[11] and Debian[12] Planet brought me even closer to the community. I received positive feedback on #dri-devel channel (IRC), and it was a way to get known to other developers of the community that works for different worldwide companies. However, I did not share every thought there because I worried with the audience. During the project, an outsider started to work on the same things that I had scheduled for my project. Initially, I decided to share knowledge in a discussion between this newcomer and a subsystem maintainer. We started a thought-building process, looking for the proper solution. However, at some point, I felt the person stole my proposed solution when I suddenly saw that solution in a patch with no mention of my authorship or co-development. After sharing this episode with others, I realized that sometimes two people do not know they are working on the same solution, and it is acceptable. However, it is not uncommon for someone to " steal " a patch in development by another. Respecting the work of others is a good community practice. A developer can use tags to grant credits when more than one works on a solution. The Linux kernel documentation describes the standards tags[13].

Contributions for VKMS driver / DRM subsystem

1st Version Date	Commit message	Developer(s)	Kind of change	# Versions	Status
July 10, 2020	drm/vkms: change the max cursor width/height	Melissa Wen	Increase test coverage	1	Merged
July 14, 2020	drm/vkms: add wait_for_vblanks in atomic_commit_tail	Melissa Wen	Fix operational behavior	1	Declined
July 22, 2020	drm/vkms: add missing drm_crtc_vblank_put to the get/put pair on flush	Melissa Wen	Fix operational behavior	1	Declined
July 30, 2020	drm/vkms: fix xrgb on compute crc	Melissa Wen	Bug-fix	1	Merged
Aug. 1, 2020	drm/vkms: guarantee vblank when capturing crc	Melissa Wen	Fix operating behavior	3	Merged
Aug. 19, 2020	drm/vkms: add alpha-premultiplied color blending	Melissa Wen	Add new feature	2	Merged

Table D.3: *Contributions sent to DRM subsystem during GSoC project*

[11] https://planet.freedesktop.org/

[12] https://planet.debian.org/

[13] https://www.kernel.org/doc/html/latest/process/submitting-patches.html#when-to-use-acked-by-cc-and-co-developed-by

Contributions for IGT GPU tools

1st Version Date	Commit message	Developer(s)	Kind of change	# Versions	Status
June 25, 2020	test/kms_cursor_crc: release old pipe_crc before create a new one	Melissa Wen	Bug-fix	2	Merged
June 25, 2020	lib/igt_fb: change comments with fd description	Melissa Wen	Improve Maintainability	2	Merged
June 25, 2020	test/kms_cursor_crc: update subtests descriptions and some comments	Melissa Wen	Documentation	2	Merged
July 6, 2020	lib/igt_fb: remove extra parameters from igt_put_cairo_ctx	Melissa Wen	Improve Maintainability	1	Merged
Aug. 26, 2020	tests/kms_cursor_crc: refactoring cursor-alpha subtests	Melissa Wen	New feature	2	Any

Table D.4: *Contributions sent to IGT GPU Tools project during GSoC*

D.1.3 Role: Independent Linux kernel developer

The Linux DRM subsystem has a very active community on IRC. Comparing with the IIO subsystem, the DRI community is larger. While the #dri-devel channel never stops to receive messages, no single message arrives in #linux-iio. Graphic developers use the IRC to ask for technical advice, patch review or acks, synchronize maintainership work, and discuss the use of tools and processor to share knowledge.

Last Version Date	Commit message	Developer(s)
Jan. 21, 2021	drm/vkms: Annotate vblank timer	Daniel Vetter
Jan. 01, 2021	drm/vkms: Decouple config data for configfs	Sumera Priyadarsini and Daniel Vetter
Oct. 13, 2020	drm/vkms: Switch to shmem helpers	Daniel Vetter
Oct. 09, 2020	drm/vkms: fbdev emulation support	Daniel Vetter
Oct. 06, 2020	drm/vkms: update todo	Melissa Wen
Sep. 27, 2020	drm/vgem: validate vgem_device before exit operations	Melissa Wen
Sep. 23, 2020	drm/vkms: Introduce GEM object functions	Thomas Zimmermann
Sep. 23, 2020	drm/vgem: Introduce GEM object functions	Thomas Zimmermann
Aug. 30, 2020	drm/vkms: Introduces writeback support	Rodrigo Siqueira
Aug. 27, 2020	drm/vkms: avoid warning in vkms_get_vblank_timestamp	Sidong Yang

Table D.5: *Contributions to DRM - Coding, Reviewing and Testing*

D.1.4 Role: VKMS Driver maintainer

Graphic maintainers on the DRM subsystem should follow a workflow to apply patches in the subsystem repository. *"The tools and workflows for maintaining and contributing to the Linux kernel DRM subsystem's drm-misc and drm-intel repositories* are documented and available publicly on https://drm.pages.freedesktop.org/maintainer-tools/index.html Any DRM community member, even more, maintainer should follow the freedesktop.org and kernel.org Code of Conduct.

"Abuse of commit rights, like engaging in commit fights or willfully pushing patches that violate the documented merge criteria or process, will also be handled through the Code of Conduct enforcement process. Violations may lead to temporary or permanent account or commit rights suspension according to freedesktop.org umbrella rules." Also, driver maintainers should be attentive to the mailing-list and inbox to catch contributions, questions, and code changes that affect the related driver.

D.1.5 Role: Co-mentor for Internship program

In this cycle, I co-mentored an intern in the Linux kernel community for the Outreachy Program. *"Outreachy provides remote internships. Outreachy internships are open to applicants around the world. Interns work remotely. Interns are not required to move. Interns work with experienced mentors from open source communities. Outreachy internship projects may include programming, user experience, documentation, illustration, graphical design, data science, project marketing, user advocacy, or community event planning. Outreachy's goal is to increase diversity in open source."*[14]

The program has two phases: the application period and the internship period. Preselected applicants start to contribute to the community they want to work in the internship in the application period. *"Prospective interns (applicants) work with mentors to complete small contributions to the project during the six-week application period."* If selected, interns work for three months to develop the internship project. Besides the internship project development, interns should participate in biweekly chats and write posts in a blog sharing their program experience. The program timeline details the tasks[15].

As a mentor, I also have responsibilities and time commitment in both phases[16]. The application phase gave me some interesting insights on newcomers.

1. For the Linux kernel project, applicants should start to set up the development environment and get familiar with the submitting patch steps before the application period starts. Applicants who started to figure out how to contribute to the Linux kernel community only when in the application period seemed to face barriers hard to overcome in the timebox of six-week. For the project that I mentored, three expressed interest, one gave up without any submission, one did not have enough time to build the project schedule, and only one was accepted.

2. The community members start to observe negative behavior even in the application

[14] https://www.outreachy.org/
[15] https://www.outreachy.org/docs/internship/
[16] https://www.outreachy.org/mentor/#mentor

phase. As the intern will work together with the community, good social skills and respecting others' work make a difference to be or not selected. I experienced some occurrence of "social faults", I saw the community coordinator's reaction and captured the opinion of other community members regarding the episode. For some long-term developers, certain behaviors show disengagement from the project, which conflicts with mentoring motivation.

3. The kernel community advises applicants to add a prefix on patches related to the Outreachy program. Therefore, community members can identify the contributions of newcomers from this program. It may help to avoid rude comments and do not happens in the Google Summer of Code program.

4. In this phase, we try to evaluate the ability to design a good project, engagement, self-commitment, and communication in more than technical skills.

The benefits of mentoring an intern are twofold: recruitment of new developers to the community and updating and strengthening subsystem knowledge. This experience exposes that even long-term experienced developers and maintainers do not know everything under their maintainership. Also, the Linux kernel solutions are collaborative work that requires developers with different expertise and backgrounds. Maintainers have responsibilities as a team leader: clerical tasks, development and people management, reviewing contributions, and connecting people.

References

[P. Adams and Capiluppi 2009] Paul Adams and Andrea Capiluppi. "Bridging the gap between agile and free software approaches: the impact of sprinting". In: *International Journal of Open Source Software and Processes (IJOSSP)* 1 (Jan. 2009), pp. 58–71 (cit. on p. 15).

[Almeida Biolchini et al. 2007] Jorge Calmon de Almeida Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, Tayana Uchôa Conte, and Guilherme Horta Travassos. "Scientific research ontology to support systematic review in software engineering". In: *Advanced Engineering Informatics* 21.2 (2007). Ontology of Systems and Software Engineering; Techniques to Support Collaborative Engineering Environments, pp. 133–151 (cit. on pp. 21, 29, 58).

[R. J. Adams et al. 2017] R. J. Adams, P. Smart, and A. S. Huff. "Shades of grey: guidelines for working with the grey literature in systematic reviews for management and organizational studies". In: *International Journal of Management Reviews* 19.4 (2017) (cit. on pp. 32, 36).

[Avelino et al. 2018] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. "Measuring and analyzing code authorship in 1 + 118 open source projects". In: vol. 176. 2018 (cit. on pp. 61, 65, 67, 68, 75–77).

[Aniche et al. 2019] Mauricio Aniche, Joseph W. Yoder, and Fabio Kon. "Current challenges in practical object-oriented software design". In: *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 113–116 (cit. on p. 19).

[Bagherzadeh et al. 2018] Mojtaba Bagherzadeh et al. "Analyzing a decade of linux system calls". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018 (cit. on pp. 61, 68).

[Bailey et al. 2007] J. Bailey et al. "Evidence relating to object-oriented software design: a survey". In: *First International Symposium on Empirical Software Engineering and Measurement*. 2007 (cit. on pp. 22, 82).

[Barr et al. 2012] Earl T. Barr et al. “Cohesive and isolated development with branches”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7212 LNCS. 2012, pp. 316–331 (cit. on p. 61).

[Budgen and Brereton 2006] David Budgen and Pearl Brereton. “Performing systematic literature reviews in software engineering”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. Shanghai, China: ACM, 2006, pp. 1051–1052. isbn: 1-59593-375-1 (cit. on p. 21).

[Beck et al. 2010] Kent Beck, M Beedle, A Bennekum, et al. “Manifesto for agile software development. agile alliance (2001)”. In: *Retrieved June* 14 (2010) (cit. on p. 15).

[Bovet and Cesati 2005] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O’Reilly Media, Inc.", 2005 (cit. on p. 15).

[Berger et al. 2014] Thorsten Berger et al. “Variability mechanisms in software ecosystems”. In: *Information and Software Technology* 56.11 (2014), pp. 1520–1535 (cit. on p. 61).

[Bettenburg et al. 2015] Nicolas Bettenburg, Ahmed E. Hassan, Bram Adams, and Daniel M. German. “Management of community contributions: a case study on the android and linux software ecosystems”. In: *Empirical Software Engineering* 20.1 (Feb. 2015), pp. 252–289 (cit. on pp. 61, 67, 68, 76).

[Brodkin 2013] Jon Brodkin. *Linus Torvalds defends his right to shame Linux kernel developers*. July 2013. url: https://arstechnica.com/information-technology/2013/07/linus-torvalds-defends-his-right-to-shame-linux-kernel-developers/ (visited on 04/19/2021) (cit. on p. 52).

[Capiluppi and Izquierdo-Cortázar 2013] Andrea Capiluppi and Daniel Izquierdo-Cortázar. “Effort estimation of FLOSS projects: a study of the linux kernel”. In: *Empirical Software Engineering* 18 (2013), pp. 60–88 (cit. on pp. 61, 66, 77).

[Corbet and Kroah-Hartman 2017] Jonathan Corbet and Greg Kroah-Hartman. *Linux Kernel Development Report - 2017*. Apr. 2017. url: https://www.linuxfoundation.org/wp-content/uploads/linux-kernel-report-2017.pdf (visited on 04/19/2021) (cit. on pp. 49, 54).

[Clark 2016] Libby Clark. *Linus Torvalds Reflects on 25 Years of Linux*. Aug. 2016. url: https://www.linux.com/news/linus-torvalds-reflects-25-years-linux/ (visited on 04/19/2021) (cit. on p. 52).

[Capiluppi, Lago, et al. 2003] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. “Characteristics of open source projects”. In: *Software Maintenance and Reengineer-*

ing, 2003. Proceedings. Seventh European Conference on. IEEE. 2003, pp. 317–327 (cit. on pp. 14, 15).

[Cohen 2018] Noam Cohen. *After Years of Abusive E-mails, the Creator of Linux Steps Aside*. Sept. 2018. URL: https://www.newyorker.com/science/elements/after-years-of-abusive-e-mails-the-creator-of-linux-steps-aside (visited on 04/19/2021) (cit. on p. 53).

[Community 2017] KernelNewbies: Community. *Scheduler*. Dec. 2017. URL: https://kernelnewbies.org/Documentation/Subsystems (visited on 04/19/2021) (cit. on p. 17).

[Corbet 2008] Jonathan Corbet. *How to Participate in the Linux Community: A Guide To The Kernel Development Process*. Aug. 2008. URL: http://www.static.linuxfound.org/sites/lfcorp/files/How-Participate-Linux-Community_0.pdf (visited on 04/19/2021) (cit. on pp. 17, 39).

[Corbet 2010] Jonathan Corbet. *On the scalability of Linus*. July 2010. URL: https://lwn.net/Articles/393694/ (visited on 04/19/2021) (cit. on p. 53).

[Corbet 2014] Jonathan Corbet. *What makes Linus happy (or not) ?* Aug. 2014. URL: https://lwn.net/Articles/608950/ (visited on 04/19/2021) (cit. on p. 47).

[Corbet 2016a] Jonathan Corbet. 25 *Years of Linux — so far*. Aug. 2016. URL: https://lwn.net/Articles/698042/ (visited on 04/19/2021) (cit. on pp. 45, 49, 77).

[Corbet 2016b] Jonathan Corbet. *A slow path to a fast fix*. Mar. 2016. URL: https://lwn.net/Articles/681062/ (visited on 04/19/2021) (cit. on p. 52).

[Corbet 2016c] Jonathan Corbet. *Group maintainership models*. Nov. 2016. URL: https://lwn.net/Articles/705228/ (visited on 04/19/2021) (cit. on p. 48).

[Corbet 2016d] Jonathan Corbet. *How 4.4's patches got to the mainline*. Jan. 2016. URL: https://lwn.net/Articles/670209/ (visited on 04/19/2021) (cit. on p. 49).

[Corbet 2016e] Jonathan Corbet. *On Linux kernel maintainer scalability*. Oct. 2016. URL: https://lwn.net/Articles/703005/ (visited on 04/19/2021) (cit. on p. 76).

[Corbet 2017a] Jonathan Corbet. *Patch flow into the mainline for* 4.14. Oct. 2017. URL: https://lwn.net/Articles/737093/ (visited on 04/19/2021) (cit. on pp. 45, 49, 73).

[Corbet 2017b] Jonathan Corbet. *The state of Linus*. Nov. 2017. URL: https://lwn.net/Articles/738230/ (visited on 04/19/2021) (cit. on pp. 48, 51).

[Corbet 2018a] Jonathan Corbet. *Code, conflict, and conduct*. Sept. 2018. URL: https://lwn.net/Articles/765108/ (visited on 04/19/2021) (cit. on p. 53).

[Corbet 2018b] Jonathan Corbet. *The code of conduct at the Maintainers Summit*. Oct. 2018. URL: https://lwn.net/Articles/769117/ (visited on 04/19/2021) (cit. on p. 52).

[Corbet 2018c] Jonathan Corbet. *Two perspectives on the maintainer relationship*. Mar. 2018. URL: https://lwn.net/Articles/749676/ (visited on 04/19/2021) (cit. on p. 52).

[Crowston et al. 2012] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. "Free/libre open-source software development". In: *ACM Computing Surveys* 44.2 (Feb. 2012), pp. 1–35 (cit. on pp. 2, 3, 5, 14, 24).

[DiBona and Ockman 1999] C. DiBona and S. Ockman. *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media, 1999 (cit. on p. 12).

[Documentation 2019] Kernel.org: Documentation. *A guide to the Kernel Development Process*. July 2019. URL: https://www.kernel.org/doc/html/latest/process/development-process.html (visited on 04/19/2021) (cit. on pp. 17, 39).

[Edge 2011] Jake Edge. *LPC: Development model diversity*. Sept. 2011. URL: https://lwn.net/Articles/458094/ (visited on 04/19/2021) (cit. on p. 48).

[Edge 2016] Jake Edge. *On moving on from being a maintainer*. Jan. 2016. URL: https://lwn.net/Articles/670087/ (visited on 04/19/2021) (cit. on p. 47).

[Edmonds and Kennedy 2013] W. Alex Edmonds and Thomas D. Kennedy. *An applied guide to research designs : quantitative, qualitative, and mixed methods*. 2013, p. 364. ISBN: 9781483317274 (cit. on pp. 24, 26).

[Eyolfson et al. 2014] Jon Eyolfson, Lin Tan, and Patrick Lam. "Correlations between bugginess and time-based commit characteristics". In: *Empirical Software Engineering* 19 (2014), pp. 1009–1039 (cit. on pp. 61, 66, 77).

[Feitelson 2012] Dror G. Feitelson. "Perpetual development: a model of the linux kernel life cycle". In: *Journal of Systems and Software* 85 (2012), pp. 859–875 (cit. on pp. 61, 73).

[Fitzgerald 2006] Brian Fitzgerald. "The transformation of open source software". In: *MIS Quarterly* 30.3 (2006), pp. 587–598. ISSN: 02767783 (cit. on pp. 2, 14).

[França et al. 2016] B. B. N. de França, H. Jeronimo Junior, and G. H. Travassos. "Characterizing DevOps by hearing multiple voices". In: *Proceedings of the 30th Brazilian Symposium on Software Engineering*. SBES '16. Maringa, Brazil: ACM, 2016, pp. 53–62 (cit. on pp. 22, 82).

[Fogel 2017] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. Second. O'Reilly Media, Jan. 2017 (cit. on pp. 3, 5, 13, 14).

[Forrest et al. 2012] Darren Forrest, Carlos Jensen, Nitin Mohan, and Jennifer Davidson. "Exploring the role of outside organizations in free / open source software projects". In: *IFIP Advances in Information and Communication Technology*. Vol. 378 AICT. Springer New York LLC, 2012, pp. 201–215 (cit. on pp. 61, 66, 67, 83).

[Foster 2017] Dawn Foster. *What the data says about how Linux kernel developers collaborate*. Oct. 2017. URL: https://opensource.com/article/17/10/collaboration-linux-kernel (visited on 04/19/2021) (cit. on p. 48).

[F. S. Foundation 2007] Free Software Foundation. *GNU General Public License*. June 2007. URL: https://www.gnu.org/licenses/gpl.html (cit. on p. 11).

[T. L. Foundation 2010] The Linux Foundation. *How To Learn Linux From the Developers of Linux. (For Free.)* Jan. 2010. URL: https://www.linuxfoundation.org/wp-content/uploads/linux-kernel-report-2017.pdf (visited on 04/19/2021) (cit. on pp. 47, 73).

[F. S. Foundation 2017] Free Software Foundation. *Overview of the GNU System*. Apr. 2017. URL: https://www.gnu.org/gnu/gnu-history.html (cit. on p. 11).

[German et al. 2016] Daniel M. German, Bram Adams, and Ahmed E. Hassan. "Continuously mining distributed version control systems: an empirical study of how linux uses git". In: *Empirical Software Engineering* 21 (2016), pp. 260–299 (cit. on pp. 61, 64, 65, 67).

[V. Garousi et al. 2016] V. Garousi, M. Felderer, and M. V. Mäntylä. "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature". In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. Limerick, Ireland, 2016 (cit. on pp. 22, 78).

[Vahid Garousi et al. 2018] Vahid Garousi, Michael Felderer, and Mika Mantyla. "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering". In: *Information and Software Technology* (Apr. 2018) (cit. on pp. 5, 21, 23, 29, 31).

[Godin et al. 2015] Katelyn Godin, Jackie Stapleton, Sharon I. Kirkpatrick, Rhona M. Hanning, and Scott T. Leatherdale. "Applying systematic review search methods to the grey literature: a case study examining guidelines for school-based breakfast programs in canada". In: *Systematic Reviews* 4 (Oct. 2015), p. 138. ISSN: 2046-4053 (cit. on pp. 23, 29).

[Higgins and Green 2008] Julian P. T. Higgins and Sally Terry Green. "Cochrane handbook for systematic reviews of interventions". In: 2008 (cit. on pp. 21, 23, 58, 59, 89, 93).

[Hsieh and Shannon 2005] Hsiu-Fang Hsieh and Sarah E. Shannon. "Three approaches to qualitative content analysis". In: *Qualitative Health Research* 15.9 (2005) (cit. on pp. 26, 37, 62, 63, 93).

[Higgins and Thomas 2019] Julian P. T. Higgins and James Thomas, eds. *Cochrane handbook for systematic reviews of interventions*. 2nd. Wiley-Blackwell, 2019 (cit. on pp. 30, 32).

[Izquierdo and Cabot 2018] Javier Luis Cánovas Izquierdo and Jordi Cabot. "The role of foundations in open source projects". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Society*. ICSE-SEIS '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 3–12 (cit. on pp. 61, 66).

[Israeli and Feitelson 2010] Ayelet Israeli and Dror G. Feitelson. "The linux kernel as a case study in software evolution". In: *Journal of Systems and Software* 83.3 (2010), pp. 485–501 (cit. on p. 61).

[Izquierdo-Cortazar et al. 2017] Daniel Izquierdo-Cortazar, Nelson Sekitoleko, Jesus M. Gonzalez-Barahona, and Lars Kurth. "Using metrics to track code review performance". In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. EASE'17. Karlskrona, Sweden: Association for Computing Machinery, 2017 (cit. on pp. 61, 66, 68, 73).

[Joblin et al. 2017] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary trends of developer coordination: a network approach". In: *Empirical Software Engineering* 22 (2017), pp. 2050–2094 (cit. on pp. 61, 65).

[Javdani Gandomani et al. 2013] Taghi Javdani Gandomani, Hazura Zulzalil, Abdul azim abdul ghani, and Abu Bakar Md Sultan. "A systematic literature review on relationship between agile methods and open source software development methodology". In: *International Review on Computers and Software* 7 (Feb. 2013) (cit. on p. 15).

[Jiang et al. 2014] Yujuan Jiang, Bram Adams, Foutse Khomh, and Daniel M. German. "Tracing back the history of commits in low-tech reviewing environments: a case study of the linux kernel". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: Association for Computing Machinery, 2014 (cit. on pp. 61, 65, 68).

[Jones 2007] Tim Jones. *Anatomia do Kernel Linux*. July 2007. url: https://www.ibm.com/developerworks/br/library/l-linux-kernel/index.html (cit. on pp. 15, 16).

[Barbara Kitchenham and Brereton 2013] Barbara Kitchenham and Pearl Brereton. "A systematic review of systematic review process research in software engineering". In: *Inf. Softw. Technol.* 55.12 (2013) (cit. on pp. 21, 22).

[B. Kitchenham and Charters 2007] B. Kitchenham and S Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. 2007 (cit. on p. 22).

[Barbara Kitchenham, Pearl Brereton, et al. 2009] Barbara Kitchenham, O. Pearl Brereton, et al. "Systematic literature reviews in software engineering – a systematic literature review". In: *Inf. Softw. Technol.* 51.1 (2009), pp. 7–15 (cit. on pp. 21, 29, 58).

[Kon et al. 2011] Fabio Kon et al. “Free and open source software development and research: opportunities for software engineering.” In: *SBES*. IEEE Computer Society, 2011, pp. 82–91. ISBN: 978-1-4577-2187-8 (cit. on p. 14).

[Lotufo et al. 2010] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Waski. “Evolution of the linux kernel variability model”. In: *Software Product Lines: Going Beyond*. 2010, pp. 136–150 (cit. on p. 61).

[Love 2010] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. ISBN: 0672329468 (cit. on p. 95).

[Lakhani and Wolf 2003] Karim Lakhani and Robert G. Wolf. “Why hackers do what they do: understanding motivation and effort in free/open source software projects”. In: *SSRN Electronic Journal* (2003) (cit. on p. 2).

[Lindberg et al. 2014] A Lindberg, X Xiao, and K Lyytinen. “Theorizing modes of open source software development”. In: *2014 47th Hawaii International Conference on System Sciences*. 2014, pp. 4568–4577 (cit. on pp. 3, 61, 65, 68).

[McManus 2016] Emily McManus. *The quotable Linus Torvalds, live onstage at TED*. Feb. 2016. URL: https://blog.ted.com/the-quotable-linus-torvalds-live-onstage-at-ted/ (cit. on p. 16).

[Melo 2013] Claudia de O. Melo. “Productivity of agile teams: an empirical evaluation of factors and monitoring processes”. In: 2013 (cit. on p. 21).

[Mockus et al. 2002] Audris Mockus, Roy T Fielding, and James D Herbsleb. “Two case studies of open source software development: apache and mozilla”. In: *ACM Trans. Softw. Eng. Methodol.* 11.3 (July 2002), pp. 309–346. ISSN: 1049-331X (cit. on p. 12).

[Miles 2015] Matthew B. Miles. *Qualitative Data Analysis + the Coding Manual for Qualitative Researchers*. 3rd ed. Sage Pubns, 2015 (cit. on p. 26).

[Osterlie and Jaccheri 2007] Thomas Osterlie and Letizia Jaccheri. “A critical review of software engineering research on open source software development”. In: *Proceeding of the 2nd AIS SIGSAND European Symposium on Systems Analysis and Design, Gdansk, Poland*. 2007 (cit. on pp. 3, 14, 19).

[Paez 2017] Arsenio Paez. “Gray literature: an important resource in systematic reviews”. In: *Journal of Evidence-Based Medicine* 10.3 (Aug. 2017), pp. 233–240. ISSN: 17565391 (cit. on pp. 22, 29, 81).

[Palix et al. 2014] Nicolas Palix et al. “Faults in linux 2.6”. In: vol. 32. New York, NY, USA: Association for Computing Machinery, 2014 (cit. on pp. 61, 68, 69).

[Raymond 1999] Eric Raymond. "The cathedral and the bazaar". In: *Knowledge, Technology and Policy* 12.3 (Sept. 1999), pp. 23–49. ISSN: 0897-1986 (cit. on pp. 1, 2, 12–14, 73).

[Riehle and Berschneider 2012] Dirk Riehle and Sebastian Berschneider. "A model of open source developer foundations". In: *IFIP Advances in Information and Communication Technology*. Vol. 378 AICT. 2012, pp. 15–28 (cit. on pp. 61, 66).

[Rigby et al. 2014] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. "Peer review on open-source software projects". In: *ACM Transactions on Software Engineering and Methodology* 23.4 (Sept. 2014), pp. 1–33 (cit. on pp. 3, 61, 64, 65, 68, 73).

[Robson and McCartan 2016] Colin Robson and Kieran McCartan. *Real world research : a resource for users of social research methods in applied settings*. 2016, p. 533. ISBN: 111874523X (cit. on p. 25).

[Rothstein et al. 2005] Hannah. Rothstein, A. J. Sutton, and Michael. Borenstein. *Publication bias in meta-analysis : prevention, assessment and adjustments*. Wiley, 2005, p. 356. ISBN: 9780470870143 (cit. on p. 22).

[Sanders 1998] J. Sanders. "Linux, open source, and software's future". In: *IEEE Software* 15.5 (1998), pp. 88–91 (cit. on p. 57).

[Sarmento 2011] Manuel Jacinto Sarmento. "O estudo de caso etnográfico em educação". In: (2011) (cit. on pp. 24–26).

[Scacchi, Feller, et al. 2006] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. "Understanding free/open source software development processes". In: *Software Process: Improvement and Practice* 11.2 (Mar. 2006), pp. 95–105. ISSN: 1077-4866 (cit. on p. 14).

[Scacchi 2010] Walt Scacchi. "The future of research in free/open source software development". In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. New York, NY, USA: ACM, 2010, pp. 315–320 (cit. on p. 20).

[Sharp et al. 2016] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. "The role of ethnographic studies in empirical software engineering". In: *IEEE Trans. Softw. Eng.* 42.8 (2016), pp. 786–804. ISSN: 0098-5589 (cit. on pp. 20, 24, 25).

[C. B. Seaman 2008] Carolyn B. Seaman. "Qualitative methods". In: *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008, pp. 35–62 (cit. on pp. 24, 25).

[C. Seaman 1999] C.B. Seaman. "Qualitative methods in empirical studies of software engineering". In: *IEEE Transactions on Software Engineering* 25.4 (1999), pp. 557–572 (cit. on p. 26).

[Shaikh and Henfridsson 2017] Maha Shaikh and Ola Henfridsson. "Governing open source software through coordination processes". In: *Information and Organization* 27 (2017), pp. 116–135 (cit. on pp. 3, 61, 64, 65, 68).

[Stallman 2018] Richard Stallman. *About the GNU Operating System*. Dec. 2018. URL: https://www.gnu.org/gnu/about-gnu.html (cit. on p. 11).

[Statt 2018] Nick Statt. *Linus Torvalds returns to Linux development with new code of conduct in place*. Oct. 2018. URL: https://www.theverge.com/2018/10/22/18011854/linus-torvalds-linux-kernel-development-return-code-of-conduct (visited on 04/19/2021) (cit. on pp. 16, 53).

[Stallman 2019] Richard Stallman. *Why Open Source misses the point of Free Software*. Apr. 2019. URL: https://www.gnu.org/philosophy/open-source-misses-the-point.html (cit. on p. 12).

[Stallman 1983] Richard Stallman. *Initial Announcement*. Sept. 1983. URL: https://www.gnu.org/gnu/initial-announcement.html (cit. on p. 11).

[Steinmacher et al. 2015] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F. Redmiles. "A systematic literature review on the barriers faced by newcomers to open source software projects". In: *Information and Software Technology* 59 (2015), pp. 67–85 (cit. on p. 14).

[Soldani et al. 2018] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: a systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232. ISSN: 0164-1212 (cit. on pp. 22, 23, 29, 31, 82).

[Scacchi and Walt 2007] Walt Scacchi and Walt. "Free/open source software development". In: *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering companion papers - ESEC-FSE companion '07*. New York, New York, USA: ACM Press, 2007, p. 459 (cit. on pp. 3, 19, 20).

[Tanenbaum 2014] Andrew S. Tanenbaum. *Modern operating systems*. 2014, p. 1101. ISBN: 013359162X (cit. on pp. 15, 16).

[Tsirakidis et al. 2009] P. Tsirakidis, F. Kobler, and H. Krcmar. "Identification of success and failure factors of two agile software development teams in an open source organization". In: *2009 Fourth IEEE International Conference on Global Software Engineering*. 2009, pp. 295–296 (cit. on p. 15).

[Tian et al. 2012] Y. Tian, J. Lawall, and D. Lo. "Identifying linux bug fixing patches". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 (cit. on pp. 61, 68, 77).

[Torvalds 2007] Linus Torvalds. *Re: Dual-Licensing Linux Kernel with GPL V2 and GPL V3*. June 2007. URL: https://marc.info/?l=linux-kernel&m=118236278730043&w=4 (cit. on p. 16).

[Torvalds 2018] Linus Torvalds. *Linux 4.19-rc4 released, an apology, and a maintainership note*. Sept. 2018. URL: https://lkml.org/lkml/2018/9/16/167 (cit. on pp. 16, 18, 53).

[Torvalds 1992] Linus Torvalds. July 1992. URL: http://www.cs.cmu.edu/~awb/linux.history.html (cit. on p. 16).

[Tan and Zhou 2019] Xin Tan and Minghui Zhou. "How to communicate when submitting patches: an empirical study of the linux kernel". In: vol. 3. CSCW. New York, NY, USA: Association for Computing Machinery, 2019 (cit. on pp. 61, 65, 66, 68, 76).

[Vetter 2017] Daniel Vetter. *Maintainers Don't Scale*. Jan. 2017. URL: https://blog.ffwll.ch/2017/01/maintainers-dont-scale.html (visited on 04/19/2021) (cit. on pp. 48, 50).

[Vetter 2018] Daniel Vetter. *Linux Kernel Maintainer Statistics*. Apr. 2018. URL: https://blog.ffwll.ch/2018/04/maintainer-statistics.html (visited on 04/19/2021) (cit. on p. 18).

[Warsta and Abrahamsson 2003] Juhani Warsta and Pekka Abrahamsson. "Is open source software development essentially an agile method". In: *Proceedings of the 3rd Workshop on Open Source Software Engineering*. 2003, pp. 143–147 (cit. on p. 15).

[Wen, Leite, et al. 2020] Melissa Wen, Leonardo Leite, Fabio Kon, and Paulo Meirelles. "Understanding FLOSS through community publications: strategies for grey literature review". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '20. Association for Computing Machinery, 2020, pp. 89–92 (cit. on p. 29).

[Wen, Siqueira, et al. 2020] Melissa Wen, Rodrigo Siqueira, et al. "Leading successful government-academia collaborations using FLOSS and agile values". In: *Journal of Systems and Software* 164 (2020), p. 110548 (cit. on p. 15).

[Zhou et al. 2017] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. "On the scalability of linux kernel maintainers' work". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017 (cit. on pp. 61, 64, 67, 68, 76).

[Zaidenberg and Khen 2015] N. J. Zaidenberg and E. Khen. "Detecting kernel vulnerabilities during the development phase". In: *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. 2015 (cit. on pp. 61, 68).