FLOSSSim: Understanding the Free/Libre Open Source Software (FLOSS) Development Process through Agent-Based Modeling

by

Nicholas Patrick Radtke

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved October 2011 by the
Graduate Supervisory Committee:

James S. Collofello, Co-Chair
Marco A. Janssen, Co-Chair
Hessam S. Sarjoughian
Hari Sundaram

ARIZONA STATE UNIVERSITY

December 2011

ABSTRACT

Free/Libre Open Source Software (FLOSS) is the product of volunteers collaborating to build software in an open, public manner. The large number of FLOSS projects, combined with the data that is inherently archived with this online process, make studying this phenomenon attractive. Some FLOSS projects are very functional, well-known, and successful, such as Linux, the Apache Web Server, and Firefox. However, for every successful FLOSS project there are 100's of projects that are unsuccessful. These projects fail to attract sufficient interest from developers and users and become inactive or abandoned before useful functionality is achieved. The goal of this research is to better understand the open source development process and gain insight into why some FLOSS projects succeed while others fail.

This dissertation presents an agent-based model of the FLOSS development process. The model is built around the concept that projects must manage to attract contributions from a limited pool of participants in order to progress. In the model developer and user agents select from a landscape of competing FLOSS projects based on perceived utility. Via the selections that are made and subsequent contributions, some projects are propelled to success while others remain stagnant and inactive.

Findings from a diverse set of empirical studies of FLOSS projects are used to formulate the model, which is then calibrated on empirical data from multiple sources of public FLOSS data. The model is able to reproduce key characteristics observed in the FLOSS domain and is capable of making accurate predictions. The model is used to gain a better understanding of the FLOSS development process, including what it means for FLOSS projects to be successful and what conditions increase the probability of project success. It is shown that FLOSS is a producer-driven process, and project factors that are important

for developers selecting projects are identified. In addition, it is shown that projects are sensitive to when core developers make contributions, and the exhibited bandwagon effects mean that some projects will be successful regardless of competing projects. Recommendations for improving software engineering in general based on the positive characteristics of FLOSS are also presented.

To my family

who have supported me on my Ph.D. journey,

that I might one day be able to return the favor,

and

to Larry Caves

who has provided a welcome and much-needed distraction

when graduate work was overwhelming.

ACKNOWLEDGMENTS

I would like to thank Dr. James Collofello, who saw potential in me as an undergraduate student completing an honors thesis. I would not be where I am today save for his encouragement to seek a higher degree and guidance thereafter. In addition to providing advice when I was unsure of the direction of my research, he has done his best to make sure that I have been financially supported throughout my graduate journey.

I am also grateful to my co-chair Dr. Marco Janssen, who was willing to take on a student outside of his field. He has introduced me to modeling and helped me navigate the social science theory and other concepts necessary for creating a high quality model but unfamiliar to a computer scientist. I appreciate the many brainstorming sessions with him that helped push me off dead center when ideas were running scarce and his general availability between regularly scheduled meetings, should the need for consultation arise.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

Free/Libre Open Source Software (FLOSS) is the product of volunteers collaborating to build software in an open, public manner. Specifically, FLOSS includes a liberal license that makes the source code available to the public, with the intent of allowing others to examine, modify, and improve the software. Although technically not a software engineering technique, FLOSS is often developed in a public and collaborative setting, relying on volunteers to make contributions to the project and often, at least to outsiders, resembling chaos. This form of developing software has been in use since the dawn of computing, when early programmers freely shared their code with others for the purpose of using and improving programs. However, the term open source software wasn't coined until 1998 [1], [2].

In order to be considered open source software, a program's license must meet certain criteria, including the following [3]:

- The source code must be available for minimal or no charge.
- Free redistribution of the software, as source code or binaries, must be permitted.
- Distribution of modified and derived works must be permitted without discrimination and under the same license as the original work.

The Open Source Initiative[1] maintains a list of licenses that meet the open source definition. As of July 2011 there are 69 verified open source licenses, excluding those that have been superceded or retired [4].

A separate but similar concept to open source software is free software. The main differences between these two categories are philosophical; those involved in the free software movement believe that providing free software is an ethical issue [5], [6] while the open source movement is interested in open development strictly for practical reasons [2]. The Free Software Foundation[2] maintains a definition for free software [6]; the criteria listed for a license to be considered free software are almost identical to the criteria included in the open source definition. Although those involved in the free software movement prefer to be treated separately from the open source movement, even the Free Software Foundation admits that most open source software is also free software and vice versa[3] [7]. Indeed, the most popular license is the GNU General Public License which, while created and endorsed by the Free Software Foundation as the preferred free software license, is also categorized as an open source license by the Open Source Initiative.

For the purpose of this document, no differentiation is made between free versus open source software. References to open source are intended to include both free and open source software unless explicitly noted otherwise. The term Free/Libre Open Source

---

[1] http://www.opensource.org

[2] http://www.fsf.org

[3] The overlap can be seen by comparing the Free Software Foundation's approved licenses (http://www.gnu.org/licenses/license-list.html) with the Open Source Initiative's list (http://www.opensource.org/licenses/alphabetical/).

Software (FLOSS) has been adopted for this document because it encompasses both types of software and is the most accepted term internationally[4]. The term "free" is meant as in "freedom", acknowledging the ability of anyone to examine and modify the source code. It does not refer to free as in price, as is commonly assumed, although most free and open source software is available without cost. The term "libre", similar to "liberated" in the English language, is included to accommodate Romance languages that do not include a base word similar to free[5]. To differentiate from FLOSS, software that does not make the source code available is referred to in this document as traditional, proprietary, or closed source.

The popularity of FLOSS is growing, and it has become common to encounter FLOSS in many situations. Examples of well-known FLOSS projects include:

- Operating systems: Linux, FreeBSD, OpenBSD, and NetBSD.
- Internet: Apache Web Server, sendmail, BIND, Mozilla, Firefox, SeaMonkey, Konqueror, Chromium, Links, and Lynx.
- Programming:
  - Web Engines: Perl and PHP.
  - Languages: Python, Ruby, and Tcl/Tk.
  - Tools: GCC, Make, Autoconf, and Automake.

---

[4]OSS, FOSS, and F/OSS are other common abbreviations used to refer to free and/or open source software.

[5]e.g., free (EN), frei (DE), and vrij (NL) versus libre (ES/FR), livre (PT), and libero (IT).

- Modeling: MASON, Repast, and Swarm.

1.1 RESEARCH MOTIVATION

Although it employs unconventional techniques, the FLOSS development process undeniably produces software that includes many positive properties. A better understanding the FLOSS development process may result in improving software engineering in general. In addition, as FLOSS has matured, more and more individuals and companies are relying on open source software. Being able to predict the future of certain components of FLOSS may benefit all the stakeholders involved, including both those who are using and developing the software, and may even lead to better software being produced through a more efficient distribution of resources (e.g., not wasting resources on inappropriate or doomed projects).

1.1.1 Positive Characteristics of FLOSS

Software engineering remains a developing field. There is an ongoing search for techniques and methodologies that can be applied to software engineering processes to improve the software produced. Building software that is high quality and reliable in reduced time are just a few of the goals for improving the software engineering process. Interestingly, many of the activities and techniques that are part of the collaborative FLOSS development process contradict traditional software engineering best practices, yet certain characteristics of FLOSS are highly desirable. Notable positive properties of software developed as FLOSS include:

- Some FLOSS projects have been shown to be of very high quality [8], [9] and to have low defect counts [10]. Indeed, Linux has been found to have a very low bug density that is less than 1/5 of the industry average for commercial software [11].

- FLOSS is able to exploit parallelism in the software engineering process, resulting in rapid development [12], [13], [14]. Unlike closed development, FLOSS is able to tap into a very large user- and developer-base, allowing design, development, bug identification, bug fixing, etc. to all occur simultaneously.

- FLOSS sometimes violates the bottleneck known as Brooks' law [15], [16], which states that "adding manpower to a late software product makes it later" [17].

- As group size increases, cooperation tends to decrease [18] and the incentive to free-ride increases, yet FLOSS development thrives on an increasing user- and developer-base [15].

- FLOSS has produced reliable, robust, portable, scalable, and complex software, and is even used in mission/safety critical applications [14], [19].

In addition, in some instances FLOSS projects have managed to successfully compete with commercial software, sometimes obtaining a greater market share than pay options, and in the process adding a level of legitimacy to a software engineering technique that is largely based on the contributions of volunteers. Examples where FLOSS successfully competes with commercial products include:

- Approximately 1/3 of one million surveyed web servers run Linux [20].

- Approximately 2/3 of web servers run the Apache Web Server [21], [22]. The next most popular server is Microsoft IIS with less than 20% of the market [21], [22], making the open source solution the most popular choice.

- Of the top 10 most reliable hosting sites, 50% run Linux and 40% run FreeBSD [23].

- The OpenDocument Format (ODF), an open standard and implementation-neutral file format for word processing, spreadsheet, presentation, and other office documents, has been adopted by many governments, including the state of Massachusetts and many countries in Europe [24]. Although not software itself, ODF was developed in an open manner and is based on the original XML file format generated by the FLOSS project OpenOffice.org[6] [25], which was also developed via an open community effort [26]. Since OpenOffice.org was one of the first projects to support ODF, some governments have also switched to using OpenOffice.org [24].

- The open source project Mozilla Firefox[7] has captured 26% of the browser market[8], making a significant dent in Microsoft Internet Explorer's market share. Google Chrome[9] has captured another 17% of the browser market[10] [27], [28], [29], [30]. Ranking as the 2nd and 3rd most popular web browsers respectively, Firefox and Chrome have eroded the market such that the once-dominant Internet Explorer, while still the most popular browser, no longer is the choice of the majority [27], [28], [29],

---

[6] http://www.OpenOffice.org

[7] http://www.mozilla.org/firefox

[8] Based on the mean values from [27], [28], [29], [30] in June 2011.

[9] http://www.google.com/chrome

[10] Although Chrome operates under a non-FLOSS license, Google has released most of the code as a separate open source project called Chromium (http://www.chromium.org). Essentially, Chromium is the development version of the browser and Chrome is Google's rebranded stable release. Sites calculating browser statistics do not appear to differentiate between the two versions of the browser, which appear to share the bulk of their codebase (Chrome includes additional features such as auto-updating, built-in plug-ins, and Google branding).

[30]; by a slim margin more people use FLOSS browsers than Internet Explorer [27], [28], [29], [30].

- BIND[11], a FLOSS Domain Name Server (DNS), accounts for 57% of DNS's in a random survey and an astounding 93% for the more stable .com, .net, and .org domains[12] [31]. By comparison, Microsoft DNS software is running on only 1% and 0.1% of the servers surveyed respectively [31].

The positive characteristics of FLOSS are appealing to software engineering in general. The fact that certain FLOSS projects are able to compete with commercial projects further speaks to the legitimacy of the FLOSS development process. Furthermore, unlike closed source development where processes and data may be highly guarded by the companies developing software, FLOSS development tends to be an open process, resulting in data being readily available for research purposes. By better understanding the FLOSS development process, it may be possible to incorporate the FLOSS practices responsible for these highly desirable traits into traditional software engineering in order to achieve similar benefits. Indeed, a report from the Workshop on Advancing the Research Agenda on Free/Open Source Software recommends that the production and organization methods utilized by FLOSS be studied in order to understand if they can be applied to other fields as well [32].

---

[11] http://www.isc.org/software/bind

[12] Timed-out requests and responses where the DNS could not be determined are omitted from these statistics.

1.1.2 Benefits of Predicting FLOSS

As open source has become a prominent player in the software market, more individuals and companies are faced with the possibility of using open source projects, which often are seen as free or low-cost solutions to software needs [33]. In the right situation, this can benefit all parties involved. For example, IBM is a company that has embraced and benefitted from FLOSS. When IBM chose to become involved in open source software, it required the company to perform an 180 degree change in thinking and policy, since IBM's business model has traditionally been to create proprietary systems that lock customers into IBM products for generations, whenever possible [34]. IBM has donated time, in the form of paid programmers, and code to the FLOSS community, including proprietary code that was previously owned and guarded by IBM [34]. Of course none of this was done strictly to benefit the FLOSS community. In return, IBM has used FLOSS as a mechanism to help unseat competitors, such as Microsoft, in markets where IBM's own products were failing [34]. For example, IBM's web server, Domino, held less than 1% of the market in 1998 while the Apache Web Server had roughly 50% [34]. Rather than fighting an uphill and losing battle to gain a foothold in the web server market, IBM joined Apache, contributing code and money to the project. While there were some initial concerns[13], three months later IBM declared it would ensure the Apache Web Server ran on all of IBM's hardware [34]. In return, IBM was able to tailor and include Apache in their own WebSphere product, which subsequently became successful [34].

---

[13]IBM was concerned about legal issues in developing Apache software [34] while the Apache community was leery of having their reputation tainted by Big Blue [34].

IBM also contributes to Linux, estimating it saves nearly one billion dollars per year by using the FLOSS operating system rather than developing its own software to meet the company's needs [34]. In essence, the cost to IBM is roughly $100 million per year [34] and the time and resources of a handful of IBM developers who are assigned to work with the Linux community. Beyond the monetary savings, IBM receives other benefits as well, such as gaining respect from the open source community [35], which likely leads to more IBM products sold. In addition, via submitting code to Linux, IBM is also able to ensure that Linux is compatible with hardware sold by IBM. Indeed, IBM advertises that all the company's servers are Linux compatible [34], a campaign the company started in 2000 [35]. Meanwhile, the Linux community benefits from greater portability of their operating system, resulting in a win-win situation.

A key to IBM's success in working with the FLOSS community is the projects the company selected to be involved in. In the case of Apache and Linux, both of these projects have evolved to be wildly successful. Linux is the dominant open source operating system, putting a dent in the commercial operating systems market[14], and Apache is the dominant web server period, even when considering commercial products [21], [22]. In 2011, it is easy to see that both of these projects are formidable forces and are likely to remain so in the future, but what about when Apache and Linux were new projects? For every successful FLOSS project there are dozens that are unsuccessful. Indeed, only 14% of projects on SourceForge, the largest repository of FLOSS projects, were updated in a

[14]Even Microsoft has cited Linux as the main competition for Windows and a viable threat [14], [19], [36].

year[15]. If IBM had instead chosen different projects and those projects shortly later failed, it would have been a waste of IBM's resources. Contributing significant resources to a soon-to-be-irrelevant project is arguably worse than not getting involved in FLOSS in the first place [37]. Indeed, the uncertainty of the survival of FLOSS projects and worries about being able to obtain support are cited as concerns by IT managers considering using open source [38].

Selecting the wrong FLOSS project to be involved with is a more significant problem for small companies than with large companies. First, small firms are more likely to adopt open source software [39] simply because they have fewer resources to begin with and therefore see the potential for larger gains[16]. In many cases, it may be impossible for a small business to write the software itself that is necessary to meet the company's needs. The company may have insufficient resources, including time, money, and/or employees with the necessary expertise. FLOSS is seen as a low-cost option [33] compared to developing a full in-house solution or buying a commercial product, assuming one is even available. If open source software with the necessary functionality already exists, a firm may free-ride. If no such software exists, it may be possible to select a close match from existing open source software and tailor it to meet the specific requirements. Either option is less resource intensive than writing the software from scratch and likely cheaper than buying a commercial solution. However, there is a significant level of risk involved in becoming involved with FLOSS, considering that most FLOSS projects fail [37]. For

---

[15]Based on the time interval from November 28, 2007 to November 27, 2008.

[16]Surveys have found that firms typically contribute less to FLOSS projects than individuals yet still receive the same benefits [40].

example, a small firm may invest in a FLOSS solution only to find the FLOSS project abandoned a year later, at which point there will be no further enhancements to or support for the project. Another iteration of selecting and enhancing a replacement project might be too costly for the business to survive. Although the stakes may be lower in the case of individuals using FLOSS, adopting an open source project and then shortly later having it fail still has negative consequences. FLOSS developers may also be interested in knowing what projects have a greater chance of survival so that the likelihood of their contributions being wasted is minimized. Even FLOSS projects, which often rely on other FLOSS projects, could benefit from accurate forcasting. Thus, the ability to predict the future of a project would be beneficial to anyone, firms and individuals alike, considering using or contributing to a FLOSS project.

Knowing the future of a project is still beneficial even if a firm only intends to use the software, but not actively contribute to it. Integrating software into a company comes with costs, including the initial selection of the software, installing and configuring the software, learning to use the software, etc. If the chosen FLOSS project becomes inactive in the future, a replacement project may need to be selected. Even if the original project fulfilled all the users' needs before it was abandoned, it may still be necessary to replace the software, partially because any form of support for the existing project has ceased to exist and partially because software and hardware are constantly evolving; sooner or later something will occur – a bug will be discovered, a computer will be upgraded to hardware that is not supported – that forces the users to find a replacement project. The costs associated with switching projects could be minimized if the future of projects could be predicted, allowing selection from a subset of projects with expected long lives. If project lifespan wasn't as

important as, say, new features and functionality being added, a subset of projects predicted to be very active could be chosen from instead.

In some cases firms don't use the FLOSS software themselves. Instead, companies sell components or services that are complementary to FLOSS. For example, Red Hat sells support services for Red Hat Enterprise Linux (previously, Red Hat Linux). Likewise, a company may market software that is complimentary and/or compatible with certain open source projects. Again, being able to predict the future of projects allows firms to better position their own marketable products and services. If a FLOSS project dies, commercial products that are linked to the FLOSS project will likely die as well.

In summary, FLOSS has a number of benefits associated with it, from the potential to develop rapidly and be very high quality, to the low cost of usage and ability to customize if necessary. However, choosing to use open source software is risky business, partly because it is unclear which FLOSS will succeed. To choose an open source project, only to find it stagnates or fails in the near future, could be disastrous. Accurate prediction of a project's likelihood to succeed/fail would therefore benefit those who choose to use or develop FLOSS, allowing more informed selection of open source projects. Unlike closed source development, where project data may be heavily guarded, data on FLOSS projects is often public, making it easier to obtain for research purposes. Better understanding the FLOSS development process may also allow all software engineering processes to be improved.

1.2 UNDERSTANDING FLOSS

The many positive aspects of FLOSS make it worth studying from a software engineering standpoint in order to better understand what causes these characteristics, so that similar techniques may be applied to all forms of software engineering. However, in reality it

is only successful FLOSS projects that exhibit these positive characteristics. In fact the majority of FLOSS projects are abandoned before developing useful functionality. Thus understanding why some projects are successful while most are not is also important. Arguably, understanding the differences between successful and unsuccessful projects may be at the crux of determining the mechanisms responsible for the positive characteristics of FLOSS.

This research is built around the simple observation that projects must receive contributions from developers in order to progress. Projects that are able to attract and retain a large number of developers have greater prospects of maturing into useful software than projects that are unattractive to the contributor community. As such, the approach taken in this research is to specifically model the project selection process of FLOSS development. Individuals working on FLOSS do not necessarily receive direct material incentives for their actions; many are volunteers donating their time, energy, and skills to software development without receiving traditional compensation. Therefore, it is critical to understand what motivates developers to choose projects in order to understand why some projects are successful. For example, are developers attracted to popular projects, projects with the most potential for reputation gain, projects which offer the ability to hone and improve one's skills, or any number of other potentially motivating factors?

To increase the understanding of FLOSS, this research uses modeling in an attempt to answer the following questions: Using publicly available data, is it possible to develop an empirically-grounded agent-based model that can explain historical patterns present in the FLOSS ecosystem? Can such a model be used to gain insight into components of the FLOSS development process, such as how developers select projects and why some projects are successful? What does it mean for a project to be successful in the FLOSS

domain? What data on FLOSS is available, what methods exist for obtaining data, and what methodological challenges exist with using the data to calibrate and validate a model? With sufficient calibration, can a model be used to accurately predict components of the FLOSS ecosystem?

Based on these questions, the main contributions of this dissertation to the field of Computer Science are:

1) Creating and validating an empirically-grounded agent-based model of the FLOSS development process.

2) Confirming existing hypotheses and qualitative data presented in literature, including:

   - Showing that FLOSS development is a producer-driven process.
   - Determining which factors are important to developers when selecting amongst projects.

3) Investigating the role of consumers in the FLOSS development domain and concluding that users exhibit only a minor influence on the FLOSS development process.

4) Determining that there are differences among proposed success metrics in the FLOSS domain.

5) Demonstrating that core developer involvement has the greatest probability of increasing the chances of project success if it occurs during the mid-stages of development.

The remainder of this document is laid out as follows: a literature review of existing FLOSS models is presented in Chapter 2, along with recommendations and justification for why FLOSS should be modeled. Existing work is compared and contrasted to this

research, highlighting where important concepts from existing models have been borrowed and incorporated into this research.

Methods for quantifying FLOSS are discussed in Chapter 3, including exploring what success means in the FLOSS domain, what factors potentially affect FLOSS, and what motivates developers to contribute to FLOSS.

A review of FLOSS data sources, including coverage of what data is typically available for projects, already existing databases, and techniques for mining data, is contained in Chapter 4.

Chapter 5 presents FLOSSSimple, a very simple, theoretical model of the FLOSS development process. FLOSSSimple is based on public goods theory and is used to explore the impact of different definitions of success in the FLOSS environment and draw some high-level conclusions about characteristics of the FLOSS development process.

A goal of this research is to demonstrate that key characteristics of the FLOSS development process can reproduced via modeling. With a well-calibrated model, it may even be possible to make predictions about the FLOSS development process. FLOSSSimple is sufficient to explore the impact of different definitions of success, but it lacks many necessary components to demonstrate reproduction of key characteristics or prediction in the FLOSS domain. Chapter 6 therefore presents FLOSSSim, an enhanced model of the FLOSS development process that includes additional components necessary to validate the model with empirical FLOSS data and produce predictions about the FLOSS process. Once FLOSSSim is calibrated and validated using publicly available data, Chapter 7 analyzes the model and uses it for prediction, to understand the importance and influence of developers and users, and to understand which factors influence the success of projects. Several scenarios are explored and directions for future work are presented.

CHAPTER 2

BACKGROUND ON EXISTING FLOSS MODELS

As FLOSS has become more popular and a formidable force in the software market, it has become important to better understand the FLOSS development process. One method to derive additional understanding of the FLOSS development process is through the use of formal models.

At a workshop for advancing FLOSS research, a group of well-known FLOSS researchers inquired if it might be possible to learn more about FLOSS, or even predict the behavior of FLOSS communities, through simulation [32]. Acknowledging that FLOSS software is sometimes as good or better than commercial software in terms of quality, functionality, and maintenance, [41] considers the economics of companies continuing to develop their own software or embracing a more open source process. [41] states that the next step should be to create analytical models of open source to explore conditions for companies adopting FLOSS techniques. [42] argues for creating generalized, quantitative models of FLOSS that could serve as prediction tools for project factors such as the success/failure of a project, the evolution of source code, the design quality, the number of developers attracted to a project, and the distribution of work on a project's components.

Due to complexity, the overwhelming amount of data, time constraints, etc., [43] suggests studying virtual communities, such as FLOSS, via models. The alternative of running real-life experiments is both costly and requires substantially high motivation from the participants [43]. Therefore, modeling is seen as a more promising and lower-risk technique to gain an understanding of virtual communities [43].

While an idea about the stability of commercial software may be obtained by analyzing the vendor, for volunteer-driven FLOSS projects predicting the stability and occurrence of future releases may be very difficult [37]. Although nontrivial to create, [37] indicates simulation might be a good option to help answer these types of questions. Indeed, simulation is needed to fully understand FLOSS due to its "inherent complexity and large heterogeneity when it comes to motives of participants, size of participation, methods of coordination, quality of output etc." [44].

Despite the need for high quality models of the FLOSS development process, there are not many researchers working on simulating FLOSS [45]. Modeling and simulating the open source development process thus remains an open research problem [46].

The following sections highlight existing work on the topic of modeling FLOSS. Section 2.1 presents a literature review of existing FLOSS models, with many models focusing on predicting aspects of FLOSS. Section 2.2 compares FLOSSSim to existing models, highlighting what makes FLOSSSim unique from earlier work. Section 2.3 contains recommendations to follow when creating FLOSS models and includes information on how FLOSSSim adheres to these recommendations when possible. Finally, the reason agent-based modeling is chosen for FLOSSSim over other techniques is discussed in Section 2.4.

2.1 EXISTING FLOSS MODELS

This section provides an overview of existing FLOSS models in order to familiarize the reader with the various FLOSS modeling efforts that have already been attempted. Concepts are borrowed from these previous research examples in developing FLOSSSim, and thus highlighting some of the different approaches to simulating and predicting FLOSS development should make it easier to understand the components and design of the model presented in Chapter 6.

It should be noted that different types of models, e.g., system dynamics, agent-based, etc., have different strengths and weaknesses and therefore the types of questions being addressed may influence the modeling technique applied. Although agent-based modeling is chosen for FLOSSSim, this section includes a review of FLOSS models regardless of the goal or type of model in order to provide a better understanding of what has occurred in the field. For an explanation of why FLOSSSim is implemented as an agent-based model, see Section 2.4.

Models have been grouped according to the modeling technique. Many, but not all, include prediction as a goal of the model.

2.1.1 Statistical Models

[47] analyzes public FLOSS data in an attempt to create a FLOSS lifespan model while identifying FLOSS lifecycle characteristics. The ratio of downloads to page views is used as an indicator for how many people become interested enough to download a project after visiting the project's homepage. In addition to downloads and page views, the number of commits and bugs reported over time are collected for projects, and the shape of the plotted data is analyzed. It is found that projects progress through up to four phases: 1) development, where views, downloads, commits, and bugs increase as rapid development and frequent releases occur, and the project receives good publicity; 2) stabilization, where commit and bug counts increase less quickly while the audience shifts to only serious users; 3) maturity, where the downloads-to-page-views ratio stabilizes, and the project mostly focuses on maintenance tasks; and 4) stagnation, where the project becomes inactive. Example projects that are in each of the phases are identified, along with some exceptions which violate the identified lifecycle patterns. This work also identifies a number of considera-

tions and caveats about the data. For example, FLOSS projects vary widely in code size, audience size, etc., making it impossible to directly compare data across projects. Phase lengths, rates of change, etc. all vary by project. However, comparisons can be made by looking at the overall shape of the plotted data. Future work includes using trend analysis and decision support methods for prediction purposes, with the possibility of seeding the model with multiple projects.

2.1.2 Machine Learning

[48] uses machine learning to predict the success of FLOSS projects. Motivated by the fact that there is no way to predict the success or failure of a project at the early stages of development, this research attempts to forecast the success of FLOSS projects by using data from the first nine months of development. Project data was mined from SourceForge using scraping tools from the FLOSSmole project[1] and then filtered to eliminate projects with fewer than seven developers[2] and fewer than 100 bug reports[3]. Projects were then hand-sorted into successful and unsuccessful categories using five selection criteria. Next, a k-means clustering algorithm was used to categorize projects into two groups (i.e., $k = 2$), where the project vectors were comprised of the following six believed-to-be-relevant factors:

1) Number of distinct email posters

---

[1]For a description of both SourceForge as a data source and FLOSSmole, as both a data source and data collection tool, see Chapter 4.

[2]This restriction was because the researchers were interested in FLOSS projects that included a team development effort.

[3]A sufficient number of bugs was required for the analysis performed.

2) Number of distinct bug reporters

3) Number of distinct bug fixers

4) Number of distinct CVS/SVN committers

5) Project outdegree[4]

6) Number of releases

The classifier was seeded with centroids calculated from three projects that were well-recognized as successful and two unsuccessful projects. The classifier correctly categorized 95% of the 42 projects into successful and unsuccessful groups, and it was demonstrated that its performance exceeded that of random classification. Reducing the number of factors necessary for prediction was then explored. The number of distinct email posters, bug reporters, bug fixers, and committers was shown to be pairwise highly correlated, indicating all four factors measure the same component of a project. Principal Component Analysis was used to systematically reduce the number of factors to the most important ones. A rerun of the classification was performed using the following revised vector components:

1) Number of distinct email posters or bug reporters or bug fixers or committers

2) Project outdegree

3) Developer outdegree[5]

---

[4]Borrowing a concept from social networking, this is the number of other projects developers on the current project have worked on and is a measurement of the centrality of the project with respect to the FLOSS developer network.

[5]The number of developers on the current project who are also involved in the development of other open source projects.

4) Number of releases

Using fewer factors, the clustering algorithm miscategorized different projects but still correctly classified 95% of the data.

[49] attempts to create rules that can predict the future of projects using data from SourceForge. First, 63 attributes were collected for a six month timespan for 55,723 projects hosted on SourceForge. Non-negative Matrix Factorization was then used to identify significant independent features, reducing the number of factors from 63 to 10. Attributes found to be the most significant through this process were number of file releases, number of developers, number of help requests, and number of opened and closed tasks. The attributes were then automatically clustered into 10 groups using the k-means algorithm ($k = 10$). Based on the automatic clustering, manual rules were then created to sort projects into their automatic categories. To test the performance of these rules, a subset of projects was manually sorted into the following categories:

- FAIL: The project was a failure, having zero developers after six months.
- TOP10: The project ranked in the top 10.
- TOP500: The project ranked in the top 500.
- NORMAL: All remaining projects belong to this category.

Support and confidence values were calculated by comparing the results of the manual rules to the automatic clusters. Support was very low for the TOP10 and TOP500 groups because the number of projects in these clusters was small. The rules had much higher support for predicting the failure of a project, with up to 78% confidence in the best case.

2.1.3 System Dynamics Models

[43] uses system dynamics to study the behavior of participants in virtual communities. While the study focuses on Comtella, a proprietary collaboration tool used by students at the University of Saskatchewan, there are some common elements that apply to other virtual communities like FLOSS, such as the inequality of contributions to the community and the temptation to free-ride[6]. The goal of the model is to provide insight into the motivation and incentive mechanisms that cause communities to develop, including the reasons why people participate in virtual communities. In particular, the model explores different levels of participation by users, where those that contribute are promoted to higher positions and offered more rewards. This is similar to a FLOSS developer hierarchy, where the continuum stretches from users, who participate at a minimal level, to core developers, who participate fully and are also awarded the ability to commit changes to the code and make design decisions, to possibly a "dictator" or leader of a project, such as Linus Torvalds of Linux, with the final say on anything related to the project. The model predicts the promotion of individuals to higher positions based on their contributions to the community. The first iteration of modeling shows the model matches relatively well to empirical data that was collected from students using Comtella.

2.1.4 Dynamic System Models

[42] points out the need for a category of FLOSS prediction models that can forecast the success/failure of projects, evolution and quality of the code, bug counts, number of programmers, etc. The authors first present a general framework and recommendations for

---

[6]For a discussion on the temptation to free-ride, see Section 3.2.1.2.2.

building dynamic simulation models of the FLOSS development process[7]; they then create and partially validate a model that follows the framework. The model is built around the principle that crowds of developers are able to decide:

- Which projects to contribute to
- Which modules to contribute to
- Which tasks to perform
- When and how often to contribute

In the model, developers may perform four actions: 1) design and implement the first version of a module; 2) fix an existing defect; 3) test the software; or 4) add functionality and improve existing code[8]. The number of developers working on a project is determined by both the profile of the developer and the quality of the software being developed. Evaluating the quality of an open source project is based on other studies' findings and is measured in the model using the increase in lines of code (LOC) between releases, the total project averages for the rate of change of LOC and number of tasks completed, and a short-term "interest boost factor" that occurs when a well-known hacker contributes to a project [50]. The model is calibrated for the Apache Web Server using data from [51], occasionally borrowing values from other studies or making educated guesses when values are not specifically available for the Apache project. The model was able to predict values three years

[7]Recommendations from the framework presented in [42] are included in Section 2.3.

[8]Although the developer preferences for each of these activities is unknown, through an interactive calibration technique it was found that the model performed best when 6.5% of the developers were interested in writing new code, 3.89% in debugging, 53.7% in testing, and 35.9% in improving functionality.

into the future[9] that matched well with the actual values of the Apache project, including LOC, defect statistics, and statistics pertaining to the four types of tasks. The model also showed super-linear growth periods at the beginning of a project's lifespan followed by reduced rates as a project matured, matching case studies of FLOSS. To demonstrate that the good model performance on the Apache project was not the exception, the authors recalibrated the model for the gtk+ project. The predicted values were not as close for gtk+ as for Apache, but were still respectable.

[52] creates a model of FLOSS based on public goods theory. Specifically, the authors model the open source phenomenon as a "game of the private provision of a public good" [52], borrowing and adapting many assumptions and results from [53], which looks at the "private provision of a public service," i.e., public services that no one wants to provide but are best created by an individual. In the model, individuals have complete information and there is no centralized control of FLOSS projects. The model is driven by strategies outlining when it is beneficial for an agent to immediately develop the software versus waiting for another agent to develop it. This includes calculating an expected lifetime utility from using the software to decide if it is advantageous to invest in developing the software now, and thus start benefiting from it immediately, or to free-ride until someone else implements the functionality, thus avoiding the cost of writing the software but also suffering from a delay that reduces the lifetime utility of the software. In addition, agents' decisions to work on a particular project are influenced by the number and quality of other volunteers' contributions. Data from developer surveys is used to validate part of the model. The model finds that FLOSS projects exhibit bandwagon dynamics; a program-

[9]The model run started in 1996 and predicted values for 1999.

mer joining a project increases the likelihood of other developers also joining. Likewise, when a developer leaves a project, the probability increases that other volunteers will also abandon the project. In addition, the model demonstrates that good programmers attract other skilled programmers and that projects with large numbers of complimentary modules are more likely to thrive in an open source environment.

2.1.5 Agent-Based Models

One of the earliest attempts at modeling FLOSS is SimCode [45], a model of developers selecting and contributing to modules within a single FLOSS project. In the model, developers' efforts are afforded to the most rewarding tasks. The model assumes that developers are more attracted to generic and low-level modules than high-level and highly specific modules. For example, writing code in the Linux kernel, which is likely to be included in many future releases and used by many users, is more rewarding than contributing code to a file system driver, which is more rewarding than contributing code to a driver for a newly released, obscure printer. Similarly, creating a new module is more rewarding than contributing to an existing module, as there is more reputation gain from being first (similar in the academic world to being the first to publish a paper). Finally, active and popular modules are more rewarding than stagnant modules, as there is a larger user base to notice the contributions. In the model, agents have perfect knowledge of all modules and contribute their individually allocated efforts to probabilistically chosen modules, where more rewarding modules have a higher likelihood of being selected. No attempt to match a module's characteristics with a developer's skills is made. The model endeavors to replicate the high Gini coefficients for the size of modules found in some FLOSS projects (e.g., Linux [54]). That is, there is a huge inequality in module sizes, with most modules being very small and

only including a single developer's code, while a few modules grow very large and contain contributions from multiple developers[10]. Although the first version of the model considers where developers focus their code-writing efforts within a project, future work may include modeling which competing projects developers select to receive their contributions.

The authors of [37] are motivated to predict aspects of FLOSS, such as the stability of a project's developer community and potential for future releases, and have created an initial model that aims to demonstrate how developers choose projects and the global effects of these choices. The model is able to successfully replicate certain known phenomena present in FLOSS development. The authors include a social networking aspect in their model and start development by collecting data from Advogato[11], FLOSS mailing lists, and FLOSS developers' blogs. The data collected are used to recreate the underlying social networks. The resulting networks are analyzed and the empirically derived network size and density parameters are included in the model. Called OSSim, the model consists of multiple projects, developers, and users. Developers and users are modeled as agents with sets of software-related problems they are interested in solving, such as desiring a text editor that can be used to modify web pages. Each of these abstract problems is then further broken down into individual features, represented as a string from all possible features.

---

[10] SimCode studies this inequality of size and developer involvement at a micro level, seen when looking at the modules within a project, but this phenomenon also appears at a macro level, when looking across a group of FLOSS projects. That is, most FLOSS projects are small and contain the work of a single developer, but a few are large and include contributions from many developers.

[11] http://advogato.org is a social networking website for FLOSS developers that allows individuals to rate FLOSS developers' abilities; essentially, FLOSS developers' work is peer reviewed and the results are published for everyone to see.

Agents select desireable projects by comparing their own problems to the features offered by projects, using Kauffman's NK model [55] to evaluate the fitness. The distribution of developers' level of skills follows data mined from Advogato. Agents learn about other projects through the social network, and if an agent becomes dissatisfied with the project he/she is currently working on, the agent will search for another project. The model was explored via two experiments. In the first experiment, two competing projects were created: one was primed with highly skilled developers while the other was assigned poorly skilled developers. The model demonstrated the intended behavior of all agents abandoning the project stocked with low skill level developers. In a second experiment, the effects of negative interactions between features developed by different developers was explored. Four projects were created and the effects and coupling between features was varied across model runs. When there were high degrees of interaction between the features, development slowed and developers more frequently changed projects. This matched expected behavior, where large, complex, and highly coupled projects tend to progress at a slower rate than small, simple projects. Although the authors' goal was to model across a large number of competing FLOSS projects, the maximum number of projects included in their published work is four – a small number compared to the actual number of FLOSS projects in reality that are competing for developers.

[56], [57], [58], [59] are a series of papers that use a combination of agent-based modeling and data mining. Specifically, these papers model FLOSS social networks, where developers and projects act as nodes and developers participating in projects form links. By understanding the characteristics of the networks, the authors hope to gain insight into why some projects are successful (e.g., examining the network characteristics around successful projects). In the model, at each time step an agent chooses to create a new project, join an

existing project, abandon a project the agent is already involved in, or do nothing, where the probability of each of these actions is based on empirical data mined from SourceForge. Network metrics, such as degree distribution, diameter, and clustering-coefficient, are then collected from the simulated network and compared to data mined from SourceForge. Modifications to the model are made and then the process is reiterated, fine tuning the model to produce networks with characteristics similar to SourceForge. [57] notes that the SourceForge network is scale-free and exhibits small world phenomenon (i.e., a high clustering coefficient and small diameter). [57] also finds that core developers keep the otherwise sparse network well-connected and these highly-connected nodes keep the degree of separation between developers low, aiding in fast communication which may help developers make well-informed decisions when choosing open source projects.

Although [60] does not focus on predicting success, the model presented does successfully reproduce several FLOSS characteristics. Using agent-based modeling, the model focuses on individuals' behavior and considers the development of FLOSS an emergent property. Software projects are modeled as collections of modules where each module has a fitness and complexity associated with it. Fitness is a measure of a module meeting the users' needs and decays over time, representing the changing needs of the users. Unlike many FLOSS models, this model includes users, who are responsible for adding new requirements to random projects. Developers also randomly move from module to module. When a developer encounters an unfulfilled requirement, he/she may choose to develop the code to meet the requirement. When a developer encounters an already-developed module, he/she may refactor the code, reducing the complexity of the module but leaving the fitness unchanged. A refactoring attempt may fail if the module is already too complex. Finally, a developer may choose to further develop an existing module, increasing the module's

fitness and complexity. If a module's fitness is already above a boredom threshold, the developer will find the work remaining on the module uninteresting and will move to another module. To validate the model, data from the four successful FLOSS projects Arla, Gaim, MPlayer, and Wine were collected. The following components were compared between the model and empirical data:

1) Size: Measured by number of functions in the source code.

2) Complexity: McCabe's cyclomatic complexity.

3) Complexity change: An indication of whether the code became more or less complex between releases.

4) Touches: The number of times a file was added, modified, or deleted.

With calibration, the model matched well with three of the empirical data, including the growth spurts and stagnation periods seen in the four selected projects. Touches did not match well. The authors found that including refactoring in the model was necessary to match the data, and users were key to causing the growth spurts through their clustering around projects. Finally, the model was sensitive to the developers' boredom threshold; when set high, high-fitness modules attracted and retained developers. When low, few projects attracted developers and those that did were eventually abandoned.

2.2 Comparison of FLOSSSim to Existing Models

Chapters 6 and 7 include a description and analysis respectively of a new model called FLOSSSim. FLOSSSim borrows components from some of the existing literature and includes unique components as well, attempting to address some of the shortcomings of existing models. Similarities to the existing models are outlined below:

- Like [47], FLOSSSim is based on projects progressing through development phases during the lifecycle of the project. FLOSSSim refers to these as development or maturity stages and includes six stages in the model rather than the four outlined in [47].

- FLOSSSim includes the concept that developers' (and potentially users') choices influence FLOSS development. Like [42], FLOSSSim is designed around developers choosing both how frequently to contribute and what projects to contribute to. As in [45], agents use probabilistic choice mechanisms to select projects based on the perceived reward or utility of each project.

- Like [37], FLOSSSim uses a mathematical abstraction of developer's problems and interests (albeit the abstraction is implemented differently between FLOSSSim and [37]) which may then be used to compare the similarity between projects and agents.

- Like many of the models, FLOSSSim is calibrated and validated using empirical data; when data is not available, estimates are used and/or searches are performed to find values that perform well.

Some of the key differences between existing models and FLOSSSim are the following:

- FLOSSSim does not use machine learning. While machine learning has been shown to be effective for predicting the future of projects [48], [49], this technique does not provide insight into the FLOSS development process itself, such as understanding why some projects are successful and others are not. In other words, machine learning acts as a black box, receiving input and producing output without providing details

of what occurs internally. Since understanding what causes success and failure in the FLOSS domain is part of the goal of this research, a white box approach is taken.

- FLOSSSim is designed around the concept of projects attracting developers. Namely, if a project can attract and retain developers, then the project will progress. Including this concept in the model means there must be a large landscape of varying projects for developers to choose from. Almost all of the existing models that allow agents to choose projects include only a small number of FLOSS projects, e.g., [45] focuses on where developers contribute within a single project and [37] considers developers choosing from up to only four projects. Yet existing literature indicates that multiple projects, including competing projects, are relevant in a model attempting to mimic real-world conditions. For example, [37] showed that developers switch to different projects based on conditions of current and competing projects. To allow for the dynamics of agents choosing from multiple, competing projects, FLOSSSim includes a large pool of FLOSS projects.

- Several models focus on the social network formed by FLOSS developers and projects, either modeling the network directly (e.g., [56], [57], [58], [59]) or including the network as a medium of communication for developers in the model (e.g., [37]). FLOSSSim does not directly include a social network component in the model.

- Many FLOSS models do not include users and none explicitly include passive users – that is, users that contribute nothing directly to the project (e.g., they do not report bugs, request new features, provide help to others in forums, etc.). FLOSSSim includes passive users with the intent of better understanding what impact, if any, this group has on open source development.

- The bulk of existing models focus heavily or exclusively on the technical factors of a project while ignoring the surrounding social factors that may impact the FLOSS development process[12]. FLOSSSim incorporates both types of factors in order to more accurately model and therefore better understand the FLOSS development process.

A summary comparing features of existing FLOSS models is contained in Table 2.1, while notable features borrowed from existing models and adapted for use in FLOSSSim are summarized in Table 2.2.

2.3 Recommendations for Modeling FLOSS

There have been a number of general recommendations for modeling FLOSS. Where possible, these recommendations are followed in FLOSSSim, thus avoiding some of the problems encountered by other researchers and benefiting from concepts that have already been peer reviewed.

It is recommended that historical data be used to calibrate FLOSS models [42]. This recommendation is followed by making extensive use of the data that is available from the FLOSS development process itself. See Chapter 4 for an overview of data sources and Sections 6.1 and 6.2.2.1 for details on calibrating the model with historical data. In addition, historical data is used for validation purposes, as outlined in Section 6.2.1.

Unlike closed source software projects, [42] notes that the number of contributors to an open source project varies widely with time and cannot be predicted. Therefore, it is recommended that models have a function to control the number of contributions based on project factors, such as developer interest in the project [42]. FLOSSSim provides this

[12]The differences between technical and social factors are outlined in Section 3.2.1.

TABLE 2.1
Comparison of existing FLOSS Models

Model	Model Type	Goal	Number of Competing Projects	Main Validation Data Source	Models Developers	Includes Passive Users
[47]	Statistical	Understand the FLOSS lifecycle	N/A	SourceForge	No	No
[48]	Machine learning	Identify factors influencing success and predict success at an early stage of development	N/A	SourceForge	Aggregate level	No
[49]	Machine learning	Identify factors influencing success and predict success	N/A	SourceForge	Aggregate level	No
[43]	System dynamics	Understand motivation, incentive mechanisms, and promotion of members in online cooperative communities	N/A	Data collected from lab experiment	Aggregate level	No
[42]	Dynamic systems	Predict LOC, defect density, etc. for a specific project	1	Data from Apache and gtk+	Aggregate level	No
[52]	Dynamic systems	Understand the dynamics of developers joining and leaving projects	More than 1	Partial validation with developer survey data	Individual level	No
[45]	Agent-based	Predict what modules developers contribute to within a single project	1	Data from Linux	Individual level	No
[37]	Agent-based	Understand how developer choose projects and the global effects of these choices	Up to 4	Conventional wisdom and knowledge from commercial software	Individual level	No
[56], [57], [58], [59]	Agent-based	Replicate and study the underlying network connecting developers and projects	Many	SourceForge	Individual level	No
[60]	Agent-based	Validate the importance of identified factors in the FLOSS development process via simulation	1	Data from Arla, Gaim, MPlayer, and Wine	Individual level	No
FLOSSSim	Agent-based	Increase understanding of the FLOSS development process and determine conditions that increase chances of project success	Many	SourceForge and developer surveys	Individual level	Yes

TABLE 2.2
Noteworthy features of existing models that are borrowed and incorporated into FLOSSSim.

Previous model	Noteworthy features	Incorporation into FLOSSSim
[47]	Work based on projects progressing through four development stages.	Projects progress through six development stages.
[42]	Designed around developers choosing preferred projects and tasks.	Developers and users choose projects based on projects' potential utility.
[45]	Agents employ probabilistic choice when selecting modules.	Agents employ probabilistic choice when selecting projects.
[37]	Mathematical abstraction of agents' needs and the ability of projects to address those needs.	Different mathematical abstraction of agents' needs and the ability of projects to address those needs.

functionality by allowing developers to choose which project(s) they will work on, where a developer's choice is based on a number of factors including the similarity between the developer's interests and the project.

Closed source software projects tend to have developers that are assigned to specific tasks. In FLOSS, this is not the case, and developers are free to pick and choose the tasks that hold their interest – or perform no tasks whatsoever if nothing appeals. [42] recommends there be a mechanism to control the number of contributions from different classes of developers, such as new, old, and core developers. FLOSSSim does not differentiate between different types of developers. However, FLOSSSim does allow developers to join and leave projects freely. In addition, developers may choose how little or much of their resources they will contribute to a project, forming a continuous gradient from fringe to core developers.

Since developers are free to choose the tasks they are interested in working on, their choice may be made based on two types of factors: factors based on the developer's profile, such as the personal interests and aptitude of the developer, and factors based on the project, such as what tasks are incomplete [42]. In FLOSSSim, developers take into account both personal and project level information when selecting which projects to contribute to.

The LOC contributed by FLOSS developers varies widely, so a model should draw from probability distributions for the amount of code contributed by each developer [42]. Instead of LOC, in FLOSSSim developers are endowed with resources that are drawn from a probability distribution. Developers may then contribute some or all of their resources to the project(s) of their choice.

FLOSS projects do not include the traditional schedules and deadlines seen in closed source software engineering. However, the amount of time to complete project deliverables has a large variance in FLOSS and thus should be drawn from a probability distribution when modeling open source development [42]. In addition, large tasks should, on average, take longer to complete than small tasks [42]. FLOSSSim respects both of these recommendations. The amount of work necessary to complete a project is drawn from a probability distribution. Developers then work on projects, with their contributions moving projects closer to completion, resulting in large projects on average taking longer than small projects to complete.

FLOSS models are likely stochastic simulations. When measuring the performance of the model, multiple runs should be averaged [42]. This is done when evaluating FLOSSSim, as described in Section 6.2.3.

In the FLOSS engineering process, volunteers develop software often without strong centralized control [13], [61]; rather, much of the code is developed in a decen-

tralized manner by individuals. Furthermore, the participants are heterogeneous, differing in their interests, needs, skills, etc. The focus on individuals and their behaviors makes agent-based modeling an excellent choice for this scenario [60]. Logical rules governing heterogeneous agents' behavior may be implemented programatically, and the FLOSS software produced becomes an emergent property of the collective action of the agents [45]. Following these recommendations, agent-based modeling is used in this research.

There exist many FLOSS projects and the existence of one FLOSS project is not completely independent of other projects. Together, all projects form a software ecosystem [46], where changes to one project may propagate, in one form or another, to other projects [62]. For example, many open source projects include other open source software as subcomponents [46]; consequently, for two linked projects a change in either project may affect the other project. As another example, there are often multiple projects trying to provide similar functionality. These projects may be viewed as competing and as such, a change in one project may affect if individuals choose this or another project instead. For this reason [46] recommends FLOSS be modeled using multiple projects, accounting for coevolution/codevelopment links between projects. Although FLOSSSim does not include explicit project-to-project links, multiple projects are included in the model along with the dynamics of agents evaluating projects with respect to one another and moving between projects based on the changing conditions.

Much of the research on FLOSS has focused on very limited groups of projects [42]. Case studies, for example, typically include only one or a few projects. While interesting qualitative results have been obtained from these studies, the conclusions reached are not necessarily globally true, or even true for most FLOSS projects [42]. For example, many case studies examine highly successful projects but ignore obsolete or failed projects,

bringing into question if the conclusions drawn from these studies can be universally applied. There is a need to move from project-specific models to more general models based on quantitative data [42]. Following this recommendation, FLOSSSim avoids using highly specific data, such as conclusions from case studies, whenever possible. When calibration data is being collected and analyzed, care is taken to use large sample populations whenever possible in order to keep the conclusions more general and applicable to a generic model.

2.4 CHOOSING A MODELING TECHNIQUE

The modeling technique chosen is based on the research goals and data availability. Important items that influenced the choice include the following:

- Based on empirical evidence and findings of existing research, the model needs to include heterogeneous attributes of the actors. Agent-based models are able to easily capture heterogeneity while other modeling techniques, such as those based on systems of equations, are more appropriate for modeling homogeneous populations.

- While quantitative data exists, much of the available data from FLOSS developer surveys is qualitative. Therefore, a modeling technique that can easily capture this qualitative information is preferred. Agent-based modeling provides a natural method to translate qualitative data into a model since rules, behaviors, etc. can be coded programatically. Some modeling techniques, such statistical models, are driven by quantitative data and ill-suited for capturing qualitative data.

- A goal of this research is to simulate phenomena that can be tested on empirical data, and to explore the consequences of different assumptions on future trends. Agent-based modeling provides a transparent, white box modeling approach that allows

changes to be made and the effects observed. Black box modeling approaches, such as machine learning, are unable to meet these goals and therefore are not used in this research.

- In the case of FLOSS, there is knowledge about the actions of individual developers based on surveys and observations, but not about the interdependencies at a global level. When low-level data is available, bottom-up modeling techniques should be chosen over top-down techniques. Agent-based models have been shown to be appropriate in situations where there is insufficient knowledge about the interactions at an aggregate level but there exists information about the low-level interactions [63].

- There exist aggregate FLOSS data that can be used to validate the model from the emergent properties of an agent-based model.

Because of these requirements and data availability, agent-based modeling is selected for this research.

2.5 CONCLUSION

The popularity of FLOSS has recently grown, and while the development process is rather unconventional, it has managed to produce very high quality software. For this reason FLOSS warrants further study in order to gain a better understanding of a different method to develop software. Many researchers have concluded that additional knowledge can be derived through modeling FLOSS. A limited number of attempts to model different aspects of FLOSS have already occurred, but modeling, and predicting, FLOSS remains an open research question. The research presented in this dissertation borrows components from existing FLOSS models, along with adding new concepts, to create a new FLOSS model that

addresses some of the shortcomings of existing models. Agent-based modeling is chosen as the technique that best fits the research being performed. Best practices and recommendations from literature in regards to modeling FLOSS are included whenever possible when designing FLOSSSim.

CHAPTER 3

QUANTIFYING FLOSS

The focus of this research is to better understand the FLOSS development process, with a particular interest in gaining insight into why some projects succeed while others fail. This chapter provides information about FLOSS necessary to reach this goal. Section 3.1 considers what it means for an open source project to be successful, noting that traditional software success metrics may not apply and thus new definitions of success must be created. Section 3.2 examines a set of factors that may influence the success of a project. Finally, Section 3.3 briefly discusses developer motivations, noting that this research aims to better understand how developers choose which projects to join.

3.1 MEASURING SUCCESS

Being able to predict which FLOSS projects survive and which are abandoned clearly has utility. In order to proceed, what constitutes success in a FLOSS project must first be defined.

Inherently, FLOSS engineering is a very different process than traditional software engineering; it is not simply traditional software engineering poorly implemented [46]. [46] characterizes FLOSS as "a different, somewhat orthogonal approach to the development of software systems where much of the development activity is openly visible, development artifacts are publicly available over the Web, and generally there is no formal project management regime, budget or schedule." In fact, much of what is considered best practices in proprietary software engineering is completely ignored in the open source world (e.g.,

providing known-to-be-buggy releases of commercial software is frowned upon as doing so can substantially drive up maintenance costs while tarnishing a company's image, yet FLOSS encourages the release of incomplete, immature software). Therefore, it should come as no surprise that the set of commercial software success metrics may not be entirely applicable to FLOSS.

Section 3.1.1 outlines traditional success metrics while Section 3.1.2 introduces success metrics proposed for the FLOSS domain.

3.1.1 Traditional Software Engineering Success Metrics

The following are metrics frequently used when evaluating the success of commercial software projects:

Meets requirements specification [64]: The traditional software engineering process includes generating documents formally specifying the requirements of the system. Thus, a system's level of success can be ascertained by comparing the software functionality to the requirements specifications. FLOSS projects, on the other hand, infrequently generate specifications for requirements of the system, at least not in the sense of traditional specification documents [65]. Part of this may be because the developers of FLOSS are typically also the users [13], [52], [65], [66]; there is no incentive, and arguably less or no need, to formally capture the requirements since those who are writing the code are also the ones who conceive the requirements based on their own needs. [65] finds that FLOSS projects avoid the formal requirements elicitation, analysis, specification, validation, and management processes used in traditional software engineering, instead developing requirements through "software informalisms." While some of the informalisms may produce documentation

(e.g., discussions on mailing lists, howto guides), no single, formal specification is generated, nor are a set of specifications deposited in a central location, making this metric difficult to apply to open source projects.

Completed on schedule [64]: Proprietary software has schedules and deadlines that must be met. To remain competitive in today's fast-paced world of computers, software must be delivered on-time to the customer in order for it to be relevant and successful. Shipping a buggy or incomplete product in order to meet a deadline can, and most likely will, lead to higher costs through maintenance and support, as well as a loss of customers, who are displeased with the software, leading to a failed project. Interestingly, FLOSS avoids this problem because there are no hard deadlines [46]. Indeed, the customers are often the developers themselves [13], [52], [65], [66]. While this provides an incentive for the developers to complete tasks quickly (as [13] points out, all open source software originates from a developer's need to scratch his/her personal itch), it also provides a reason to not produce a release version of the software until it is functioning correctly. This is because the developer suffers directly from poor quality software since it is the developer him/herself who is also the user. A poor quality release may also cost the developer time and energy in providing support for the buggy software. [13] argues that the lack of deadlines gives open source a competitive edge when it comes to building quality into the product[1].

---

[1] [13] cites two strategies incorporated into FLOSS development to avoid the "deadliness of deadlines" [13] – that is, the low-quality software that results from the requirement to complete too many features by an unreasonable deadline. One option is to keep the deadline but relax the feature list. When the deadline occurs, only the features that have been fully implemented and tested are included in the release version, and no promises are made about when any particular feature will be included. The second option is to keep the

Completed on budget [64]: Although functionality is important, software must also be developed at a reasonable cost. Projects that are money sinks are not considered successful. FLOSS does not operate on a traditional budget per se [46] and therefore this metric does not apply.

Penetrates the market: A software product must be able to attract and retain a user-base. Therefore, one way to measure the success of a project is via the market share the product captures. With closed source software, the number of users can be tracked because the users are required to buy the software and/or licenses. This is not the case with FLOSS, where users may obtain the software anonymously from multiple sources, including from other users, in most cases making it impossible to know the number of people using the software or the percentage of the market captured[2].

Turns a profit [69]: In the commercial world, a software product may be considered successful if it turns a healthy profit for the owning company [69]. Obviously, this metric does not apply to open source because 1) expenditures are non-existant since most of the work and resources are donated by volunteers; few FLOSS projects have an actual

---

required feature list but relax the deadline. The next release will occur only after all the features in the list have been implemented and tested.

[2]There are a few exceptions where the market share for open source projects can be estimated [67]. One exception is the Apache Web Server. Web servers may be queried to determine what web server is running. Netcraft (http://news.netcraft.com), a company that explores, analyzes, and provides research data about the Internet [68], uses this functionality to determine the market share held by web servers by querying on the order of 200 million sites [21] and tallying the results. Other exceptions might include software that "phones home," checks for updates, includes a quality/bug feedback component, etc., which by contacting a central location provides a mechanism to count the number of copies of the software that are in use.

budget; and 2) income from the project is zero because the software is distributed for free. Without income and expenditures, using profit as a metric makes little sense.

In general, success metrics traditionally used for closed source software do not fit well when applied to open source projects.

3.1.2 Proposed FLOSS Success Metrics

Since traditional success metrics are not necessarily applicable to open source projects, a new set of success metrics must be considered. The following describes FLOSS success metrics that have been proposed and discusses how this data might be collected.

Project completion [70]: If a project manages to include all the desired functionality, it may be considered a success. In truth, this rarely occurs as most software, especially in the FLOSS world, is never complete. Even very functional software continues to exhibit scope creep as new functionality is desired, old bugs are fixed, etc. Furthermore, FLOSS projects rarely have end goals defined to begin with [71], making it impossible to gauge how close a project is to completion, and the lack of formal specifications makes FLOSS more susceptible to volatile requirements and scope creep [71]. Thus, defining success as those projects that are 100% complete will result in only a minute subset of projects being deemed successful.

Progression through development stages [66]: Since projects are rarely ever complete, evaluating a project based on its progression through development stages may be a more valid metric. SourceForge, for example, requires projects to be listed in the following self-assigned development stages (listed in order of increasing maturity): planning, pre-alpha, alpha, beta, production/stable, and mature. Success might there-

fore be defined as a threshold (e.g., projects that reach the beta stage might be considered successful) or, more dynamically, as the progression of a project through stages (e.g., occasional upgrading to a subsequent stage might indicate success while stagnation in any early stage might indicate failure of a project).

The metrics used to evaluate the performance of a project might differ depending on the project's development stage [71]. Therefore, the definition of success might differ for a project that is in, say, the alpha stage versus the production/stable stage. In this case, the development stage would act as an input in selecting the appropriate success metric, but would not be part of the metric itself.

On SourceForge, a project's development stage is viewable via the web interface. Therefore, the information for this metric can be obtained for at least SourceForge projects.

Developer satisfaction [70]: FLOSS is a product of developers. Therefore, if the creators of the software are happy with the project then arguably the project is meeting its goals and consequently can be considered successful. Indeed, interviews with developers have revealed that developers evaluate the success of projects they are working on based on their satisfaction with the project [72]. Developer satisfaction can be measured via surveys [70].

User satisfaction [66]: The level of satisfaction of users may be used as an indicator of the success of a project [66]. Unfortunately, unlike the developer community, which is a semi-well-defined group of individuals [70], the user community is poorly defined. While developers typically register with a project in order to participate, users may obtain software from multiple sources anonymously, making users impossible

to track. As a result, user satisfaction cannot be measured with traditional surveys as it is impossible to locate members of the target audience. Other non-traditional methods of obtaining this data suffer from substantial drawbacks. One option is to collect data from sites like freshmeat[3] and Ohloh[4], which allow users to voluntarily review and rate open source projects. Unfortunately, the data on these sites is provided by a non-random sample of users. Indeed, most projects receive overall high ratings with low variance, indicating only those who feel positively about a project take the time to provide feedback [70]. Messages on mailing lists can also be mined for users' opinions of a project, but again this represents a non-random sample and may be biased (e.g., users are more likely to post a message when they are expe-riencing a problem and seeking help). A final method for obtaining data on user satisfaction is to build surveys into the software itself. However, if the surveys are optional or can be bypassed, again the feedback collected likely will be biased. If the surveys are mandatory to continue to use the software, users may be interested only in the speed of completing the survey and not in providing honest feedback. Other problems associated with having the software itself collect and report usage data is further described in "Popularity with users" on page 49.

Number of developers [48], [67], [70], [73], [74]: Since FLOSS projects rely on both attracting and retaining developers in order to progress, the number of developers associated with a project has been proposed as a success metric [48], [67], [70], [73],

---

[3]http://freshmeat.net is a website that tracks new releases and updates for software, with an emphasis on FLOSS. Both SourceForge and freshmeat are owned by Geeknet.

[4]http://www.ohloh.net is a public directory of both FLOSS projects and developers.

[74]. [48] argues that the number of active developers should be increasing, or at least remain constant, in order for a project to be successful. Similarly, [73] argues that only after a project goes through a transition where it rapidly gains developers does it enter a successful phase. Often developers must first register with a forge and then join a project in order to contribute. Thus, the number of developers working on a project can be obtained by checking the number of developers registered with a project. Unfortunately, this number can be misleading as many of the registered developers may be inactive. For example, a developer may register for a project and later lose interest but never unregister with the project [48]. Since inactive developers do not cause a project to progress, obtaining a count of active developers may be a better measure of success. One option is to process a project's Software Configuration Management (SCM) logs and count the number of unique committers who have contributed in the recent past [48], [70]. However, this count will only include developers with code commit privileges, while it is often the case that people without commit permissions also contribute code to a project indirectly [75]. Analyzing recent activity in mailing lists, forums, bug lists, etc. may provide a more accurate account of those working on a project, as it will include people who contribute ideas, bug and feature requests, code snippets, etc. without being formally registered with the project [48], [70].

Instead of counting all the individuals associated with a project, a variation is to only count the core developers. As has been shown in multiple studies, the work distribution on projects is highly skewed, with a small group of core developers producing

the bulk of the contributions[5] [51], [77], [78], [79]. It has also been shown that a small number of dedicated developers are involved in multiple projects[6] [77], [80]. Thus, by counting only core developers, only the most influential members are considered when evaluating the level of success of the project. Keeping this in mind, [48] creates a new metric called project outdegree, a count of the number of other projects developers are involved with. Essentially, a larger number of dedicated developers working on a project will result in a higher project outdegree, as these core developers are likely also involved in other projects. Project outdegree can be calculated by obtaining a list of developers registered with each project and then cross referencing the lists to discover which developers work on multiple projects.

Unfortunately, the number of developers is sensitive to the size of a project – small projects likely have fewer developers than large projects – which makes it difficult to compare different size projects using this metric without some form of normalization or adjustment.

SourceForge includes the list of developers associated with a project on the project's homepage. Likewise, SCM logs can be downloaded and processed to discover active committers. The availability of mailing lists and forums archives, bug reports, etc. for

---

[5]One study found that four percent of developers contribute almost 88% and 66% of new code and code fixes respectively [76]. A different survey found that 10% of the developers were responsible for writing 72.3% of the code base, and the top 10 developers, a mere 0.08% of those surveyed, accounted for an astounding 19.8% of the code [77].

[6]One survey found that 25 developers, 0.19% of those surveyed, participated in more than 25 projects, and 250 developers, 1.9% of those surveyed, were involved in more than five projects [77]. Using data collected by [77], [80] finds that the 100 most prolific developers contributed to 1886 FLOSS projects, averaging an astounding 19 projects per developer.

further processing will depend on the project (e.g., not all SourceForge projects use the bug tracking system provided by SourceForge, forum and mailing list archives may not be public).

Popularity with users: Regardless of all other indicators, if a project is not being used it is arguably a failure. This is supported by a survey [72] which asked developers to define success for FLOSS projects. Opinions differed on criteria for determining if a project was successful, but all developers agreed that a lack of users indicated the project was a failure. Use of a project may mean direct use by end-users or the inclusion of the project as a component in other (FLOSS) projects.

Determining the popularity of a project with users is non-trivial. Unlike the developers of a project, who can be tracked to some degree via the list of programmers registered with the project or other digital trails, there is no simple or robust method to accurately determine the number of users (see "Penetrates the market" on page 43 for an explanation of why this is the case). Still, there are several proposed methods that attempt to capture the popularity of a project with users through proxies.

One possible proxy for the number of users of a project is the number of downloads [48], [66], [81]. While download counts are available from sites like SourceForge, these values under-represent the actual number of downloads, since users may choose to check out the code via anonymous reads from the project's code repository rather than downloading a stable release. Anonymous reads are tracked separately from download counts of stable release versions. In addition, open source licenses allow the software to be redistributed, so there may be additional sources beyond a project's

homepage from which the software can be obtained[7]. Furthermore, downloads may not accurately represent the number of people using the software: some individuals may download the software but choose not to use it[8]; other software is frequently used but rarely downloaded[9] [70], [82].

Another proposed metric is the amount of traffic on a project's website [64]; unfortunately, project traffic suffers from some of the same problems as download counts.

Frequency of use [66] or number of uses [69] is another proposed metric that has specific advantages. Namely, there is a difference between a project that is downloaded and used once versus a program that is downloaded and then used regularly. The former may indicate a failed project (e.g., the user tried the program, found it unacceptable, and then deleted it) while the latter may indicate success[10]. Unfortunately,

---

[7]In some cases a project may keep a web page on a popular forge simply as a placeholder that links to an official homepage hosted elsewhere. Since the software is downloaded from the non-forge site, the forge's count will incorrect. See 4.2.2.3 for more details.

[8]A variation that helps address this issue is to instead consider the ratio of downloads to page views [47]. Instead of measuring the number of people using the software, this ratio represents the number of people who, after visiting the project's homepage, believed the software to have enough utility to warrant downloading, and thus is a method of measuring the software's perceived usefulness.

[9]vim, an enhanced clone of the popular vi text editor, is an example of this. Although vim is available on essentially any UNIX installation, the software is almost always packaged with the operating system distribution and thus not downloaded as an individual project. Furthermore, vim is stable software so it is unlikely a system administrator would ever choose to upgrade the software independently of an incidental upgrade that would occur during a system upgrade.

[10]However, this may not always be true and depends on the purpose of the software. For example, software that uploads new firmware by its very nature is run infrequently; indeed, running the software once and then deleting it may be the normal use case.

this metric is also difficult to obtain. In theory, the number of times a program is executed could be recorded by the program itself and this data could occasionally be reported to a central repository. However, this requires the source code of each project to be modified and is likely to be frowned upon by many open source users due to this technique's resemblance to spyware[11]. Some open source software already asks the user for permission to gather and send information back to the project's developers (e.g., a crash handler in OpenOffice collects data to help the developers debug the scenario leading to the failure; the Debian Popularity Contest is a package Debian users can install that reports packages installed and dates that components are used, with the project aggregating this data and making it available to the public [83]).

Considering that FLOSS is an online activity, the presence of a project on the Internet may be a method to judge the project's popularity. Based on this, [69] proposes using web search engines to determine the success of a project. Comparing this to published research papers, where publications that are cited frequently are considered influential, [69] argues the same for links to projects on the web. Namely, the projects that are frequently linked to on the web are important and therefore successful. [69] finds that the number of pages with backlinks to a project's homepage corresponds reasonably with the success of the project. Since some web search engines already index backlinks, as well as allow querying for backlinks, the infrastructure to use this metric already exists.

---

[11] Avoiding spyware/malware is an incentive for using FLOSS. Since the source code is available for review, the chances of malicious code being slipped into the software unnoticed by the open source community is reduced. Furthermore, if malicious code is discovered it can be eliminated by anyone, since the source code itself is available.

Unfortunately, the popularity of a project is sensitive to the type of project, making it difficult to compare the level of success of dissimilar projects. More generalized projects (e.g., a word processor) will inherently have a larger potential audience than highly specialized projects (e.g., an Esperanto spell checker). For this reason, market penetration may be a better metric (see "Penetrates the market" on page 43 for problems associated with using market penetration with FLOSS).

Number of subscribers [67]: Some projects publish announcements and information about new releases. Counting the number of people subscribed to announcement mailing lists, RSS feeds, etc., might be used as an indicator of success since the number of subscribers is an indicator of the public's interest in the project [67]. The number of subscribers is expected to be smaller than the number of users since many users will be content to occasionally check the status of a project rather than receive frequent announcements from the project. However, the group of subscribers might include non-users who are interested in learning more about the project but have not yet chosen to use the software. Even including the non-users, the number of subscribers may better represent the group of "core" individuals interested in the project – individuals that may be anxiously awaiting a bug fix, a feature enhancement, etc., and therefore might be seen as dedicated to the project. Not all projects post announcements or provide a way for interested individuals to subscribe. Furthermore, the frequency of postings will vary from project-to-project and may have an influence on those who subscribe (e.g., too many postings may drive away potential subscribers). Finally, like the popularity of a project with users, the number of subscribers is also sensitive to the size of the project's target audience. Therefore, it may be difficult to compare projects using this metric.

Activity level [43], [48], [64], [66], [69], [70], [74]: Possibly more important than the number of developers working on a project is the amount of work that is actually occurring. For this reason, the level of activity on a project may be an indicator of success [43], [48], [64], [66], [69], [70], [74]. Activity may include code commits, changes in lines of code, opened and closed bug reports, opened and closed feature requests, mailing list and forum posts, official releases, etc. It might be argued that even if minimal code is being written, a community that is active in supporting users of the software signifies a successful project. The activity of a project can be determined by processing logs, e.g., SCM or bug report logs.

An example of a hybrid activity ranking is available from SourceForge. For each project on the site an activity index is calculated based on a project's commits, bug reports and feature request, time since the last file release, time since a project administrator logged in, etc. [84]. A list of all hosted projects, ordered by decreasing activity value, is provided presumably to help those browsing for software to select active and vibrant projects. freshmeat provides a similar metric called a vitality score, which is based on the number of project announcements, date the project was created, and date since the last version was released [85].

Turnaround time to fix bugs [48], [64], [70] and implement features [70]: The amount of time it takes to fix bugs and/or implement new features may be indicative of the health of a FLOSS project. Furthermore, the turnaround time to fix bugs and implement new features has high variance among projects, indicating this metric may be a good choice to differentiate among projects [70]. [48] argues that successful projects fix bugs quickly. This measure can be further enhanced by taking into ac-

count the severity of the bugs[12] and importance of the features requested[13]. Instead of using turnaround time, another option is to measure the proportion of bugs/features that have been fixed/implemented [70], again possibly taking into account the severity/importance of the bugs/features. Most projects use software to track bug status and feature requests so these data may be available.

Release frequency [48], [64], [66], [67], [70], [72], [74]: [13] provides the guideline to "release early, release often" in order to foster success in open source software development, providing examples where this school of thought has created thriving projects. Failure to produce a release version of software most likely is a sign of a failed project. Extending this notion, failure to produce a new version within a certain timeframe may also indicate the project is defunct[14]. Unfortunately, defining a reasonable timeframe may be difficult, and some rapidly progressing projects rarely, if ever, produce release versions, preferring instead to keep the development version stable and available for download (e.g., stable nightly builds). Release frequency can be considered a coarse-grained activity measure. File release lists can be obtained for projects hosted by SourceForge.

---

[12]Bugs are often assigned a priority based on their severity. Ideally, critical bugs are assigned high priorities and fixed quickly.

[13]Some projects allow the users to vote on the importance of new features as well as the urgency of fixing bugs [70].

[14]In a study it was found that projects that have not created a release within the last year are typically abandoned [72].

Ports [70]: If a project receives many requests to port the software to other systems, it may be an indicator of success. By requesting ports, users are acknowledging the utility of the software over anything currently available on other systems. SourceForge lists metadata on projects, including what systems the software will run on. As a metric, porting does not make sense to certain types of projects, e.g., projects written in JAVA are inherently platform independent and therefore cannot be ported.

User involvement [70]: Part of the efficiency and quality of FLOSS comes from users' involvement in the development process. For example, users test the software and provide valuable feedback in the form of bug reports and feature requests. Without users providing this information, FLOSS development becomes just a group of developers creating software and loses some of the key advantages it has over closed source development. Therefore, the number of users actively engaged in the software improvement process may help measure the success of a project. User involvement may be partially calculated by comparing multiple logs (e.g., people who have filed bug reports but are not included in the SCM logs likely are users and not developers for the project). This metric is sensitive to the type of project, making it difficult to compare projects with different target audiences.

Project recognition [70]: In a survey [70], developers mentioned public recognition as a measure of a project's success. This may include links to the project on the web or possibly connections to and/or influence of the project on other projects, including both commercial and FLOSS software. In some cases it may be possible to measure recognition, e.g., links to a project may be discovered with the help of search engines, as outlined in "Popularity with users" on page 49.

Note that many of the success metrics listed are sensitive to project size. For example, large projects will more often than not have more developers, more activity, etc., than small projects. Therefore, it may be necessary to perform some form of normalization when comparing dissimilar projects.

To effectively measure the success of FLOSS, it may be beneficial to use multiple metrics in order to provide a more well-rounded measurement [70]. While many of the proposed metrics have been used in research, there is no agreed upon subset that acts as a standard for measuring open source success [70], [75]. For example, [66] considered projects that were used more frequently, in advanced development stages, and had more activity to be successful while [48] took into account project activity, release frequency, and the links between a project and other open source projects.

Section 7.1.4.1 describes using FLOSSSim to explore the effects of using different success metrics.

3.2 Factors Influencing FLOSS Project Success

What factors exist that influence the success of FLOSS projects? Do certain practices improve the chances of a project being successful? Fortunately, much research has been conducted to explore questions about the correlation between certain antecedents and a project's success. This section enumerates factors and summarizes the findings of the research. The goal is to identify influential factors and incorporate them into the model.

3.2.1 Types of Factors

Factors affecting FLOSS projects include both technical and social factors.

3.2.1.1 Technical Factors

Technical factors are aspects that relate directly to a project and its development and are typically both objective and easy to measure. Examples of technical factors include lines of code, number of developers, and other project attributes.

The online nature of open source development means that technical factors are often automatically collected by the software tools used in the development process. Sites that provide tools for FLOSS development may make these data available to the public, and thus readily available to researchers. SourceForge, for example, makes much of the project data it tracks available via a web interface. A sample, non-exhaustive list of technical factors that can be obtained from SourceForge is shown in Table 3.1. For additional information on technical factor data sources and data extraction techniques, please see Chapter 4.

3.2.1.2 Social Factors

Social factors pertain to aspects that personally motivate or discourage individuals from engaging in open source development/use. Examples of social factors include reputation from working on a project, matching interests between a project and the developer/user, popularity of a project with other developers/users, and perceived importance of the code being written (e.g., core versus fringe development [45]). Most social factors are subjective and rather difficult, if not impossible, to measure. Despite this, it is hard to deny that these might influence the success/failure of a project and therefore social factors are considered in the model. Fortunately, the social factors being considered fall under the domain of public goods, for which there is already a large body of work published (e.g., [86], [87], [88], [89], [90]).

TABLE 3.1
Non-exhaustive sample of technical factors available from SourceForge on a per project basis.

Factor	Description
Page views	The number of project pages that have been served.
Hits to project logo	Number of times a project's logo has been served. Note that this is different than page views; a project hosted elsewhere will not affect SourceForge's page view count but may affect the project logo hit count if the offsite web pages link to the SourceForge-hosted project logo.
Downloads	The number of times each release file has been downloaded from SourceForge.
Bytes served	The amount of data served on behalf of a project.
Rank	Project list ordered by amount of activity, number of downloads, number of page views, number of project logo hits, number of forum posts, or amount of tracker activity.
Bugs	List and status of opened and closed bug reports.
Feature requests	List and status of opened and closed feature requests.
Support requests	List and status of support requests.
Patches	List of patches.
Forum posts	Messages posted to any project forums.
SCM activity	CVS, SVN, etc. commits, anonymous reads, etc.
Developers	List of developers registered with the project.
Administrators	List of developers registered as administrators for the project.
Registration date	Date the project was registered with SourceForge.
License	Open source license used by the project.
Intended audience	Target users for the software, e.g., advanced end-users, system administrators, etc.
Release date	Date of the most recent release version of the software.
Topic	Type or category of software such as games, browsers, networking, etc.
Operating system	The operating system(s) on which the software will run.
Translations	Languages available.
Development status	The maturity stage of the software, selected from: planning, pre-alpha, alpha, beta, production/stable, mature, and inactive.
User interface	User interface, e.g., Win32, Plugins, Java Swing, X Window System, Command line, etc.
Programming language	Programming language(s) used by the project.

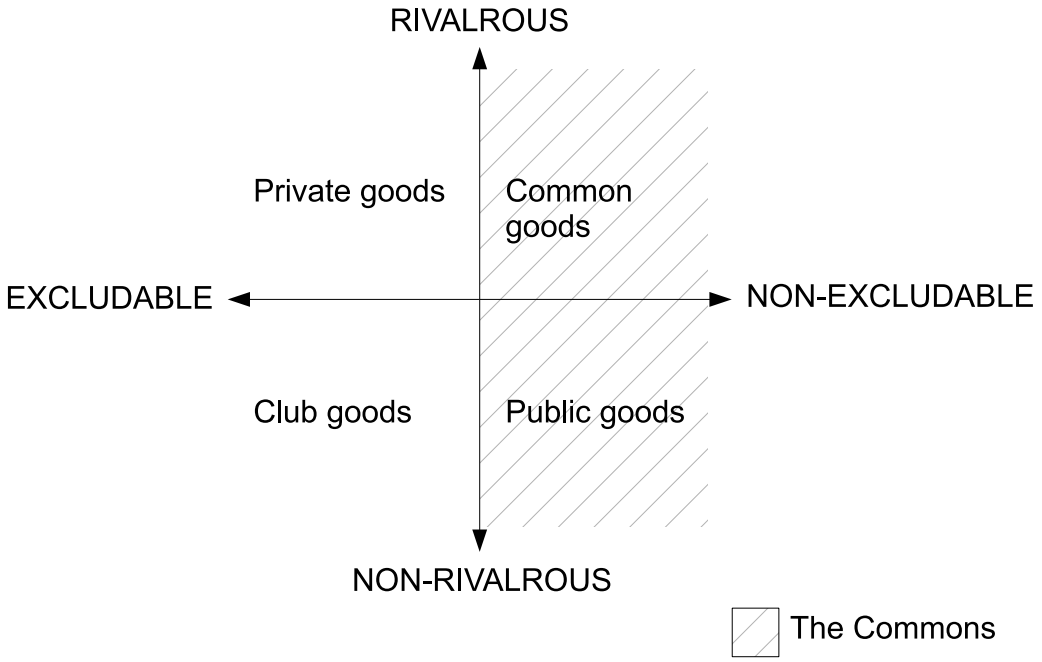


Fig. 3.1. Types of goods based on rivalry and excludability. FLOSS is considered a public good because it is non-rivalrous and non-excludable.

3.2.1.2.1 FLOSS as a Public Good: In economics, goods are divided into four categories based on whether the goods are rivalrous and excludable. Rivalry means that the good can be used up or consumed; the use of a rivalrous good by one individual leaves less of the good remaining for others. Excludablity refers to whether others can be stopped from using the good. Figure 3.1 illustrates the four types of goods by plotting rivalry and excludability on separate axes.

Private goods are rivalrous and excludable. Most items that can be bought at a store are in this category. For example, food is rivalrous in that an individual buying food leaves less food for others to buy. Food is also excludable in that by eating it, others are unable to also consume it.

Club goods, also called toll goods, are non-rivalrous and excludable. Examples of club goods include movie theaters and zoos. Only those who pay or are members of a club

may use the good. However, one person enjoying the good does not leave less for others to enjoy.

Common goods, also called common-pool resources, are rivalrous and non-excludable. Common examples are natural resources, such as clean fresh water or fish in the ocean. No one can be excluded from catching fish, but each fish caught is one fewer for others to catch.

Finally, public goods are both non-rivalrous and non-excludable. Examples include non-encrypted broadcast radio signals and light from a streetlight. Everyone is able to freely enjoy these goods and one person's use does not reduce the quantity of good for others.

FLOSS is a public good [12], [52], [91], [92], [93] and may be considered part of a new class sometimes referred to as digital public goods[15] [91], [94]. It is non-excludable in the sense that the source code is typically available to anyone for free[16] [17], often posted on a website for uncontrolled download. The fact that software is a digital, reproducible good makes it non-rivalrous; a person, for example, downloading and using a program does not affect another person's ability to download and use the same software. Arguably

---

[15]Another example of a digital public good is Wikipedia (http://www.wikipedia.org), where the product is encyclopedia articles instead of software.

[16]Technically, the Open Source Initiative's definition of open source allows distributors to charge a nominal distribution fee for the source code, thus making it excludable. In reality, this is almost never done. In addition, most open source licenses permit the code to be redistributed for free, so even if one party chooses to sell the code, another party in possesion of the same code may choose to redistribute it for no cost.

[17]In order to meet the Open Source Initiative's definition of open source, a project's license must not discriminate against anyone or any group, nor limit what field the software may be used in. Essentially, this guarantees that no one is excluded from obtaining and using open source software.

FLOSS is not a pure public good because the licenses typically require credit be given to the developers (e.g., a developer's name appears by the code he/she contributed and, as per the license, the attribution may not be removed), whereas no part of a public good is owned by an individual. Furthermore, some open source licenses permit the code to be privatized, even allowing the code to be absorbed into commercial, closed source software products[18]. In this sense, FLOSS may be seen as sharing features with common-pool resources [95]; that is the public code may be moved into the private domain, which threatens the future availability of the source [95] for others to use and/or further develop[19].

3.2.1.2.2 The Tragedy of the Commons: Goods that are non-excludable are sometimes referred to as the Commons, as shown in Figure 3.1. These goods are collectively owned and anyone from the collective is therefore free to use them. This results in two potential problems. First, in the case of public goods, a characteristic emerges known as the collective action problem [96]. Essentially, the concern is that individuals are rational beings that are interested in maximizing their own utility. If an individual can have a good for free, there is no incentive for that individual to contribute to or maintain that good. If all individuals behave this way, there is the danger that the good is never created in the first place and thus no one benefits. Individuals that use goods without contributing are known as free-riders.

The second problem is known as the Tragedy of the Commons [97] and is traditionally applied to common-pool resources. Since the use of common goods is subtractive, a

---

[18] See Section 3.2.2.1 for a description of different types of open source licensing.

[19] A form of licensing known as a copyleft is employed by many FLOSS projects to stop this from occurring. See Section 3.2.2.1 for a description of copylefts.

problem occurs when too many people use a good. The classic example is a common parcel of land on which cattle may feed. It is to each individual's benefit to continue adding cows to the land even if this exceeds what the land can support. The risk is that if too many cows are added, the grass will be exhausted and the field will no longer be able to support any cows. The problem occurs because the benefit to the individual – gaining another cow – is large while the damage to the resource is spread across all users of the field. The Tragedy of the Commons occurs with many natural resources.

Although the Tragedy of the Commons originally applied to rivalrous goods, [72] extends the definition to FLOSS, arguing tragedy in FLOSS is "when collective action ceases before a software product is produced or reaches its full potential." The logic is that abandonment amounts to the same result as overuse – partially developed software that did not reach a level of useful functionality is no longer useful to the users, just like an overgrazed field is not useful to ranchers raising cattle. Arguably the tragedy is bigger in FLOSS in the sense that an overgrazed field likely produces some cattle for each rancher before being completely exhausted while a FLOSS project that is abandoned before useful functionality is reached provides no utility to users.

Although there has been much research on the Commons and public goods, most of this work is not specific to FLOSS; for example, some of the research explores why people volunteer to contribute to public goods and what contextual factors increase these contributions. The findings of this literature are applied when designing the model, as are findings from publications investigating how FLOSS works, extensive surveys of developers asking why they participate in FLOSS (e.g., [1], [98]), and comments and opinions of FLOSS users (e.g., [38]).

3.2.2 Existing Research on Factors

To better understand what affects the performance of FLOSS development, physical attributes (e.g., programming language, software architecture design, content management system), community attributes (e.g., degree of user involvement, leadership characteristics, social capital), and institutional design (e.g., norms, formal rules, governing structure) should be considered [99]. A number of studies have already considered various factors and their impact on the success of open source projects. These studies can be further subdivided into two subcategories: those that look at factors that directly affect a project's success (e.g., [64], [81], [100]) and those that look at factors that attract developers to a project, thus indirectly affecting the success of a project (e.g., [13], [15], [52], [101]). For the purpose of this research, both categories of factors are considered. These potential factors, along with the findings of the studies, are presented in the following subsections.

3.2.2.1 Licensing

Licensing is at the crux of FLOSS; the license is, in fact, what makes software open source. However, there are a large number of licenses, with the Open Source Initiative certifying 69 licenses as Open Source Definition compliant[20] [4]. Do the differences in the licenses affect the success of a project, for example by changing the attractiveness of the project to developers/users, or are all open source licenses essentially the same?

A particular type of open source license is also known as a copyleft. Whereas traditional copyrights protect the rights of the person who created the software by prohibiting others from copying, modifying, or redistributing a product, copylefts use copyright law

---

[20]As of July 2011. This number does not include those licenses which have been superceded or retired but also meet the Open Source Definition.

to extend these same rights to the general public. Copylefts further require that all derived works be released under an equivalent license – often the exact same license – thus ensuring that any code derived from copylefted software will remain open to the public. Also known as a reciprocal license due to this requirement, this has resulted in copylefts earning the nickname of viral licenses since the license spreads like a virus to all derivatives of copyleft protected software. The GNU General Public License (GPL), the most popular open source license[21], is the most well-known example of a viral license.

At the other end of the spectrum from copylefts are permissive free software licenses. Unlike copylefts, which prohibit any additional licensing restrictions from being added when releasing modified software, permissive licenses may allow the addition of more restrictive terms than that of the original license when software is re-released. For example, BSD-style licenses are a popular example of permissive licenses that allow the code to be used in essentially any manner, even to be closed and used in proprietary software released under a non-open source license; the only stipulation is that the programmers of the open source code be acknowledged.

Numerous other open source licenses exist somewhere between the reciprocal licenses and the permissive licenses. Licenses on the former end of the spectrum are sometimes (somewhat confusingly) referred to as restrictive licenses, because they include terms (i.e., restrictions) that require the software stay open source. Software on the latter end of

---

[21] On Sept. 12, 2010, http://sourceforge.net/softwaremap/?&fq[] lists almost 43% of the hosted projects as using the GPL and just over 7% using the related and very similar Lesser General Public License (LGPL). The next most popular license is the BSD License, which is used by only 5% of the projects on SourceForge. Other studies have put the percent of projects using the GPL as high as 72% [64], [101].

the spectrum are sometimes called unrestrictive or nonrestrictive licenses, since the terms allow the software to be used in any manner, provided credit is given to the authors.

Viral licensing allows software written under other licenses to be absorbed by a viral license – the caveat being that the original project cannot benefit from modifications/improvements that occur because of the requirement that the changed version adopt the viral license. This occurs, for example, when combining code from two different projects, where one employs a viral license and the other does not. The resulting software must be released under the viral license in order to satisfy the viral license's terms. This is a point of contention with open source developers, as this practice seems to violate the spirit of open source, namely by not allowing improvements to be contributed directly back to the original project so others can benefit from them as well. Modifications may still be returned to the open source community, but they will now fall under the viral license.

It is extremely common for FLOSS projects to include other FLOSS projects as dependencies. For example, an open source web browser may rely on an open source library to handle decompressing JPEG's. Two licenses are said to be compatible if code under each license can be mixed and/or combined to create new software. Essentially, this means that none of the terms and conditions of the two licenses conflict with one another.

Since the GPL is so popular, of particular interest is if other licenses are GPL compatible. If a project fails to use a GPL compatible license, it may be at a serious disadvantage because this significantly reduces the number of other open source projects that it is possible to collaborate with. Interestingly, other licenses may be compatible with the GPL, but once combined with code that uses the GPL, due to the viral nature of the license, that code must always be released under the GPL. When mixing code with different compatible licenses, the fact that the GPL "takes over" any other license has upset some open source developers.

Arguments abound about which is the more free, less restrictive license. The Free Software Foundation, which created and maintains the GPL, argues that the GPL is less restrictive because it forces the software to remain public and open. Meanwhile, those in favor of permissive licenses point out that when combining with the GPL, software using permissive licenses loses freedoms, such as the ability to be modified and released as closed source. Thus, the argument is that the GPL takes away some of the freedoms originally afforded to the software and therefore it is actually more restrictive. For this reason some open source licenses have added specific terms aimed at disallowing the code to be released under the GPL, thus protecting a different notion of "free".

The importance of licenses and license compatibility can be seen when creating a large project from smaller projects. Through its choice of licenses, a project immediately affects what other projects might be interested in using it and restricts what other projects are available for it to use. Developers looking to add functionality to their project may be forced to select a less desirable project – or worse yet, write the code themselves from scratch – if the best project's license is incompatible with their own project. This may result in duplicated efforts by developers working on different projects – effort that could have been spent on improving quality or adding other functionality. For this reason, it is advantageous to adopt a popular license, or at least a license that is compatible with other popular licenses, if any form of collaboration is going to occur.

It can be seen that licensing in the open source domain is a huge, complex, and important topic. There are efforts to reduce the number of licenses and avoid the creation of new ones unless absolutely necessary in order to simplify the complications associated with interactions between licenses and even different versions of the same license [102], [103]. Full coverage of issues regarding license compatibility is beyond the scope of this

document. An overview has been provided to demonstrate that license choice may indeed have an impact on a FLOSS project.

Reputation is often cited as a reason developers contribute to software[22]. That is, developers may be interested in contributing where their work will bring them the largest reputation boost, and licensing may be seen as affecting the potential gain. For example, licenses that allow code to be absorbed into commercial products may be viewed as undesirable since the commercial product may be in direct competition with the open source version, resulting in a reduced audience to notice the contributions of the developers [64]. By requiring that source code stays open, restrictive licenses help ensure that even small contributions have at least the potential for long-term benefits [104]. Perhaps the worst-case scenario would be open source software that was developed under a permissive license, only to be privatized, improved, and released as commercial software. In this case, the very same developers who created the original open version may end up buying the commercial version, yet they will not have the ability to modify the software to meet their specific needs [64]. Other motivations for contributing to open source also seem to favor a copyleft license. For example, developers who are looking to improve their programming skills[23] need the code to remain open in order to obtain feedback and learn from other expert programmers. Likewise, a programmer looking to increase his/her desirability to a potential

---

[22] Although reputation is frequently cited as a reason developers contribute to open source, actually only 9.1% of developers in a survey cited reputation as a reason for joining the open source community [1].

[23] 78.9% of developers indicate learning and developing new skills is a reason to join the open source community [1].

employer[24] may want to show his/her code as being included in a stable release, which again requires the code remain in the open source domain.

Keeping in mind that the popularity of a project may be linked to a project's level of success, [64] considers the effects licensing has on the number of users. It is shown that projects using nonrestrictive licenses accumulate more subscribers to project announcement lists than projects using restrictive licenses [64]. This seems to indicate that users find more utility in projects with flexible licenses.

Licenses likely play a part in the motivation of developers [15]. Unlike users, open source developers may have an inherent interest in keeping the code they write in the open source domain. If developers are more attracted to projects which ensure this through the employment of restrictive licenses, one would expect these projects to have more development activity. However, it has been found that the effect of license type (i.e., restrictive versus nonrestrictive) on the number of software releases is statistically insignificant [64]. Likewise, [105] analyzed FLOSS projects in the healthcare industry but found that license restrictiveness did not affect the probability of projects being classified as successful[25]. Another study found that the average output per contributor is significantly higher for projects with nonrestrictive licenses [104]. Further investigation into this finding showed that projects with restrictive licenses tend to have larger numbers of developers, many of whom contribute little and thus drag down the average [104]. The larger number of devel-

---

[24] 23.9% of developers indicate improving their job opportunities is a reason to join the open source community [1].

[25] In this case, success was based on project activity, project downloads, SourceForge rank, and number of participants [105].

opers associated with restrictive license projects may indicate an ideological motivation of open source developers [104].

[106] considers the effect licensing schemes have on the efficiency of a project[26]. While this is not a direct measure of success, arguably the efficiency of a project does impact if a project will succeed; terribly inefficient projects, for example, will likely burn out before producing useful software. [106] finds there is no significant difference in project efficiency across different licenses.

Different types of FLOSS may favor different licenses. For example, projects aimed at end-users tend to favor more restrictive licenses while projects targeted towards developers generally have less restrictive licenses [101]. This analysis can be made even more specific than these two broad categories. For example, software written for commercial operating systems with a primary language of English tends to use unrestrictive licenses while games and software developed in a corporate setting tend to favor restrictive licenses [101]. Thus, it may be more complicated than certain licenses increasing the chance of software being successful; the license that will most improve the chances of success may depend on the specific type of software being written.

3.2.2.2 Organization Sponsorship

Organizations sometime choose to be involved in FLOSS projects. This is done for a variety of reasons. In some cases, a company may find an open source project almost meets their needs and choose to get involved in order to enhance the software with the additional

---

[26] [106] uses Data Envelopment Analysis (DEA) to evaluate the efficiency of transforming inputs into outputs in the context of FLOSS development. For the analysis, the number of developers and their effort are the inputs; the size (in bytes) of the code, LOC added and deleted, number of files, number of check-ins, and the development status are the outputs.

functionality the firm requires. Financially, this may make more sense than purchasing a commercial product or developing a custom solution. For example, IBM spends $100 million per year to pay a set of employees to work on the Linux operating system [34], estimating that it saves $900 million per year over developing its own in-house operating system to meet the company's needs [34].

Organizations may choose to be involved with FLOSS to capitalize on positive aspects of open source, such as innovation and speed of development [107], [108]. For example, the company formerly known as Sun Microsystems (now absorbed by Oracle) maintains two versions of their office suite: OpenOffice is a FLOSS version while StarOffice is a commercial version. Sun pays developers to work on the open source version, benefiting from the contributions of the community as well. Sun then uses the code from OpenOffice as the base for releases of StarOffice.

Organizations may also become involved with FLOSS in order to widen their user-base. A hardware company, for example, may be interested in having their products supported by open source software. By doing so, a whole section of the market becomes available that otherwise might not be. A hardware manufacturer may even be able to corner the market if their product is the only option that has decent support in the open source community, and one way to increase the probability of this occurring is to work with the FLOSS community, possibly providing hardware specifications or even donating code to certain projects. IBM employs this strategy by making sure Linux runs on all of the servers they sell [34], [35]. Companies may also donate resources to a project in order to gain respect and build goodwill with the FLOSS community [35], [107]. Essentially, a firm's involvement in FLOSS may be an opportunity to advertise and build public relations, with the hopes that open source community members will think favorably of the company the next

time they are in the market for certain products. Even a minimal investment may have significant payback; as pointed out by an employee of a company involved in FLOSS, "When a company sponsors an open source project they need to realize that people are going to perceive [that the company is]...far more involved in [the project] than it might actually be," [109].

Companies may sell products or services that are complimentary to FLOSS [107], [108]. For example, a firm might sell training or support for a certain FLOSS project; it is therefore in the company's best interest to make sure the project remains active and relevant, possibly by paying developers to work on the project.

Sometimes companies choose to sponsor open source projects in an attempt to challenge a dominating standard [108]. For example, businesses may sponsor Linux in an attempt to unseat Microsoft's dominance of the operating systems market[27] [108]. In doing so, companies are able to create a competitive project at a relatively low cost [108]. Furthermore, if the sponsored project employs a restrictive license, there is no concern that it will be privatized in the future and turn back into the same situation which was originally being addressed [108]. Similarly, companies may choose to sponsor projects that create/support open standards [107], especially if a company has products that also support the standard and there are threats of other firms creating incompatible, proprietary, or competing standards [107].

Occasionally a firm may attempt to sabotage a FLOSS project which is perceived as a threat. In some cases a company's association with a project alone may be enough

---

[27] A prime example of this is IBM, which jumped to support Linux after its own operating system, OS/2, failed [34].

to derail the project [67], [93]. In other cases, the firm may endeavor to gain control or influence over the project in order to encourage decisions that are advantageous to the firm but contrary to the best interests of the project and/or FLOSS community. A legal example of sabotage occurred when, starting in 2003, SCO filed a series of lawsuits against various entities, claiming it owned the copyright on code used in UNIX and that the same code, which appeared in Linux, was illegal. SCO claimed that anyone using Linux was therefore in violation of copyright law. It was revealed that Microsoft, whose dominance of the operating systems market with Windows was (and still is) being eroded by Linux, was instrumental in both encouraging and funding the SCO lawsuits [110]. The strategy to eliminate Linux backfired; the lawsuits further fueled the open source community's hate for Microsoft, and although not all lawsuits are yet resolved, in general the rulings have been against SCO[28].

The influence of a firm's involvement on the success of a project has the potential to be negative or positive. One of the main principles of the Free Software Foundation is that all software should be free, as in liberated (i.e., free to use and modify for any purpose) [6]. Indeed, 37.9% of respondents in a developer survey indicated the belief that "software should not be a proprietary good" was a reason to stay in the FLOSS community [1]. In the same survey, 28.9% of the developers indicated they continue to work on FLOSS "to limit the power of large software companies" [1]. In some cases, being anti-Microsoft serves as motivation to participate in FLOSS [15]. Thus a company's involvement may actually taint a project and serve as a disincentive for developers to be involved, especially for those

[28]In SCO vs. Novell, the court ruled that Novell, not SCO, owned the copyrights to UNIX [111]. Novell has indicated they have no interest in suing over UNIX [112] and do not believe Linux violates the copyrights they own [112].

who view for-profit organizations as counter to the open source culture and ideals [67], [93]. In the most extreme cases, companies may jockey to put their own employees in the key positions of a FLOSS project, gaining control or even hijacking a project [113]. Inherently, there may be conflicts of interest and goals between a business, which is seeking to maximize its own profit, versus the FLOSS community, which has other widespread motivations for creating software [113]. If a company is successful in gaining control, developers from the FLOSS community may abandon the project [114].

On the other hand, if an organization does not taint or try to control an open source project, sponsorship may have a positive effect. An organization that pays developers to work on a project or donates other resources may increase the vitality of the project [67]. Sponsorship may provide benefits to the project that otherwise wouldn't be possible. For example, IBM's involvement with Linux ensures that Linux is compatible with all IBM hardware. Linux is also frequently run on servers. With IBM's involvement, the Linux developer community is able to be proactive and ensure that there will be no problems with new IBM servers [34], rather than being reactive and fixing the problem after the servers are installed and in use by the general public.

Studies on the effects of organization sponsorship vary. [115] finds that projects with firm sponsorship tend to be in more advanced development stages and more frequently reach the production/stable stage. Projects with company involvement also tend to be larger and more active [115]. [67] finds that sponsored projects increase in popularity significantly more than unsponsored projects. However, [105] finds project sponsorship is not useful in predicting software success.

3.2.2.3 Target Audience

Target audience inherently is tied to the type of project. For example, office applications appeal to a different segment of the population than network utilities. At a coarse level, target audience can be split into two categories: developers and end-users. The majority of the research done on target audience has used these two categories.

Most software is born from a developer scratching his/her own personal itch [13]. A developer may therefore be motivated to join projects that are interesting and can help solve the developer's own problems [52], [93]. Consequently, projects targeted at developers appear to have an advantage simply because the people who have a problem are also the ones with the skills to write the software solution. Furthermore, projects that address common problems may also be at an advantage; the more common the problem, the larger the segment of the population that will develop the same itch and attempt to scratch it. Thus software addressing frequently occurring needs may have a larger body of developers interesting in writing the software solutions than software aimed at highly specific or unusual tasks.

Projects targeted at developers also appear to have an advantage when considering the user base. With the occasional exception, open source is not known for being user friendly; FLOSS projects often lack proper or up-to-date documentation [65], [116] and may require a high level of computer skills to install and use [116]. End-users may not possess these skills, while many developers will. Thus, projects aimed at developers might be more popular, albeit with skilled users, which may influence the success of the project.

In general, there are more projects aimed at developers than any other category of target audience [117]. The class of projects aimed at developers is also the most rapidly growing category of open source projects [117].

One study found that FLOSS projects on topics of interest to developers and system administrators were more successful than projects on topics of interest to end-users [66]. Interestingly, the same study also found there were more projects on topics of interest to developers and system administrators than to end-users [66]. In addition, developer-focused projects have been shown to increase in popularity more than end-user projects [67].

On the other hand, projects that listed developers as the intended audience were no more successful than projects that listed end-users as the target audience [66]. It has also been shown that the vitality of a project is not affected by the project's target audience [67]. Finally, the efficiency of a project, where inefficient projects may be less likely to succeed, is not affected by the intended audience [106].

3.2.2.4 Governance and Coordination

[13] famously characterizes traditional software engineering as resembling a cathedral and FLOSS development techniques as mimicking a great babbling bazaar. Amazingly, out of the chaos of differing ideas, agendas, and approaches materializes stable, useable, and sometimes high quality open source software [13]. The fact that FLOSS development violates many of traditional software engineering's best practices, including the organizational structure, makes open source of great interest to those intent on improving traditional software development processes.

Contrary to the initial appearance of FLOSS development being total chaos, there does exist some form of coordination hierarchy even if it is not explicitly defined. In general, at the top of the chain of command are project leaders and core developers. This group performs the bulk of the work [51], [77], [78], [79]. Below this are central developers, who semi-regularly contribute to the project, followed by peripheral developers, who contribute infrequently. If the hierarchy is extended to users, next are active users, who provide con-

tributions in the form of testing the software, reporting bugs, submitting feature requests, etc. At the bottom of the hierarchy are passive users, who only use the software but do not provide any direct contributions to the project themselves[29]. The number of core developers is an order of magnitude smaller than the number of central and peripheral developers, which is an order of magnitude smaller than the number of users [51], [76].

At the top level, there are three types of governance employed by FLOSS projects: benevolent dictatorships[30], rotating dictatorships[31], and boards of directors[32] [119], [120].

The coordination techniques employed by a project have the potential to have a huge impact on the project. A project that cannot effectively coordinate the efforts of its volunteers will likely never create a usable software product. In addition, because FLOSS is volunteer driven, the developers must be kept happy or they will abandon the project, or possibly open source development altogether. Developers in general do not like to be bogged down in heavy weight processes, especially when such processes provide little or no return on investment. In short, developers who enjoy writing code would prefer to maximize their time programming and minimize the time spent on other activities related to the process.

---

[29]The hierarchical command structure categories outlined here are based on [118].

[30]Linux employees this form of governance, with Linus Torvalds having final say on everything about the project [119]. Emacs is another example of benevolent dictatorship [119].

[31]This form of governance is used by Perl [119].

[32]This form of governance is used by Apache [119], Mozilla [120], and FreeBSD [120].

Despite the fact that governance and coordination techniques may significantly impact a project, very little work has been done to study the effects of coordination and governing structure on project success. [120] considers the continuum from control to anarchy in Mozilla and FreeBSD. While closed source software relies on "diligent project management," it is found that FLOSS developers prefer minimal control and flat control structures [120]. In addition, open source developers dislike explicit rules, commands, and centralized government, and would rather rely on norms, self-organization, and individual autonomy [120], [121]. A balance, therefore, must be found between control and anarchy to increase a project's chance of success. Control, such as required code approval processes and rigid commit procedures, is thought to increase the quality of the software [120] and is a key component of software process improvement [122]. However, tight control, such as freezing the code for long periods of time while performing testing or building release candidates, will slow down the development process [120] and possibly eliminate other advantages FLOSS development has over traditional software engineering. Long freeze periods may also cause contributors to lose motivation, as it has been shown that a key motivational factor for developers is the ability to quickly see the results of their work [123]. Anarchy, on the hand, is supposedly necessary to attract and retain volunteers [120]. Thus both control and anarchy seem necessary for a project to be successful, yet these two components conflict with one another.

In addition, there is a question as to the impact certain highly ranked developers have on the success of a project. For example, Linus Torvalds is responsible for piecing together releases of Linux based on thousands of contributions that are made to the project. Indeed, Torvalds admits that his job is mostly about guiding the project and controlling quality [34]. Is Torvalds easily replaceable, with other developers able to as competently

and skillfully perform the same job, or are his abilities in managing his project extraordinary, making him key in propelling the project towards success? It has been found that projects with highly rated administrators tend to be more successful [66]. Similarly, [124] argues that the personalities and attitudes of those involved may affect the popularity of a project.

3.2.2.5 Documentation

Open source projects that lack proper documentation appear to be at a severe disadvantage. From a user standpoint, no matter how functional or high quality the software may be, if the user cannot figure out how to install and use it, the project has little value. Beyond users, documentation is also important to developers, especially developers interested in volunteering for a project. Proper documentation can lead to a better overall understanding of the project, which helps new developers get up to speed faster and may even increase the quality of developers' contributions [81]. Likewise, documentation can lead to increased maintainability [81]. Therefore, documentation may tie to a project's popularity, both with developers and users, and thus may affect a project's success.

Since documentation appears to be crucial to a project, one would expect open source projects to include, at minimum, sufficient levels of documentation to install and use the software for common tasks. The time spent writing documentation is expected to have a high return on investment; simply put, without documentation no one, save those involved in the project, may be able to run the software, meaning the effort invested in the project is largely wasted since the project is unable to reach its full potential, user base, or developer base. In reality, many projects suffer from minimal or out-of-date documentation [65], [116].

In a study of successful and unsuccessful projects it was found that more projects include user documentation than developer documentation [81]. However, no link was found between either form of documentation being available and the success of a project [81]. Furthermore, [81] argues that developer documentation is not necessary since competent developers are able to learn through project observation, e.g., examining the source code to learn coding standards. Another study found that projects using wikis, which sometimes serve as a form of project documentation, were no more efficient[33] than those without wikis [75].

3.2.2.6 Systematic Testing

The goal of testing is to increase the reliability and quality of software by identifying bugs (and subsequently fixing the bugs), preferably shortly after they are introduced, when they are potentially easier to resolve. Both users and developers may be interested in quality software; whether or not a project engages in certain forms of testing may impact the quality and thus the interest in the project. Testing may be manual or automated.

Some projects use rigorous testing methods to rapidly discover introduced errors. For example, some FLOSS projects use tinderboxes. A tinderbox is a computer that continuously and automatically downloads the latest version of the software, builds it, runs a sets of tests, and reports the results [120]. A single project may have multiple tinderboxes so that the software can be tested on different hardware, operating systems, etc. [120]. Some projects perform frequent smoke tests, possibly in addition using tinderboxes. Mozilla, for

---

[33] [75] uses DEA to evaluate the efficiency of transforming inputs into outputs in the context of FLOSS development. For the analysis, the number of developers and years of existence are the inputs; the number of downloads, web hits, LOC, development status, and size, in bytes, of the most recent source release packages are the outputs.

example, closes the development tree to changes daily while the code is tested, and the tree is not reopened for changes until all tests on all platforms pass [120].

There also may be control mechanisms in place to maximize the quality of code contributions. For example, a gatekeeper may need to first be convinced a contributor's code is functional before allowing it to be committed to the tree [120]. Through this process, the code may need to first pass a set of tests (e.g., unit tests, integration tests) before it is even considered for adding to the source tree.

Projects may also have release management procedures that incorporate forms of testing. For example, some projects will produce release candidates. Essentially, release candidates are versions of the software that are beyond beta testing but may still have some minor bugs. Creating a release candidate is an invitation to the community to test the software as though it was a stable version. If bugs of sufficient concern are found, they will be rapidly fixed and a new release candidate made available. If a release candidate is found to be free of severe bugs, it is upgraded to a release version. Using release candidates allows a project to take advantage of a large testing audience before declaring the software stable – a testing strategy often employed by FLOSS projects that is rarely used in closed source development. Prior to creating a release candidate, projects may first go through a more rigorous testing procedure than the regular day-to-day testing [120] in preparation for creating the release candidate. Some projects may even freeze the code for months in advance of releasing a stable version and focus only on testing and fixing bugs during this time [120].

A final consideration of the testing process is how a project tracks issues. Some projects may have comprehensive plans for tracking, prioritizing, and fixing bugs. Other projects may rely on more ad-hoc methods of managing bugs.

It has been found that bug tracking is more common in successful projects [81], but projects that use Tracker, SourceForge's issue tracking tool, are not more efficient than projects that do not use Tracker [75]. Successful projects more frequently produce release candidates prior to stable releases [81]. However, forms of automated testing are just as likely to be used in successful and unsuccessful projects [81].

3.2.2.7 Quality

Quality may be an important factor when individuals select projects. Low quality projects inherently appear to have lower utility and therefore may be rejected when considering projects, whereas high quality projects, even if they are difficult to initially set up and understand, may still offer sufficient return on investment to make the effort worthwhile. The quality of software may be measured via various means, and the quality metrics applied may differ from individual to individual. For example, quality may be evaluated based on number of defects, reliability, performance, useability, design, etc.

Using a survey, it has been shown that a FLOSS project's quality has a positive effect on both its use and user satisfaction [125]. In addition, as user satisfaction increased so did use of the FLOSS software [125]. Interestingly, the same study considered the quality of service offered by the community surrounding a project (e.g., the project's developers and other users who might offer support). High quality support from the community did not lead to increased use of the project [125].

[125] makes no attempt to understand the effect quality has on contributors to projects. However, since the developers of FLOSS are often also the users [13], [52], [65], [66], and users are drawn to quality, it is likely that quality also has a positive impact on attracting developers.

3.2.2.8 Programming Language

An advantage of FLOSS over closed source development is the ability to take advantage of a larger pool of talent. In the case of a company developing software, there are always more smart people outside the company than there are in the company [34]. FLOSS isn't limited this way, and indeed Linus' Law [13], which states that "Given enough eyeballs, all bugs are shallow," is a major argument for why open source development not only works but in some cases outperforms traditional software development.

Programming language may be an important factor in influencing success because it immediately limits the number of developers that are eligible to work on a project. The size of the subset of eligible developers may vary widely depending on the programming language(s) used. Projects that use common languages, such as JAVA or C/C++, have access to a larger percentage of the developer population than those that use uncommon or unpopular languages, for which there may only be a handful of developers involved in the FLOSS community. More developers means more resources to develop software, as well as access to more talent. Thus a project using an uncommon programming language may also be limiting the size of developer pool. A smaller pool may mean not only that there are fewer developers to potentially join the project but also that less talent is available to tap into. In addition, there may be fierce competition among projects for these developers because of their scarcity.

It is possible that programming language also has an effect on the size of the user base. For example, projects that use interpreted languages, such as JAVA or Perl, may require more skills from the users in order to setup the environment in which the software will run. Even projects that use non-interpreted languages, such as C, may still require substantial computer skills (e.g., resolving dependencies, installing libraries) in order to compile

them. Note that it is common practice for FLOSS projects to provide release versions of the software in the form of source code only and require the users to build the software on their own computer. Some projects may release binaries for specific platforms, but the source code is still available for those using systems not supported by one of the binaries. Projects that release precompiled binaries are less likely to have the size of their user base affected based on the programming language used. In other words, the amount of effort required to install and run the software affects the size of the user base; programming language and distribution method both impact the level of skill needed to get the software running and thus also affect the number of users.

It has been found that projects using common programming languages are more active, reach more mature development stages, and are more frequently used [66]. In addition, [105] finds that the programming language used by a project is useful for predicting the success of a project.

3.2.2.9 Target Operating System

Similar to programming language, the target operating system of a project affects the number of developers eligible to work on the project. Obviously, if a programmer does not have access to the operating system used by a project, he/she will not be eligible to participate in that project. However, developer preferences may also play a role. Not surprisingly, the overwhelming majority of FLOSS developers prefer to use FLOSS operating systems, with only 2.2% favoring Windows [1]. Thus while the default operating system for commercial software is Windows, open source projects targeted at open source operating systems may, surprisingly, be at an advantage. Some projects are platform independent (e.g., projects written in JAVA) and thus operating system does not have an effect on the number of eligible developers.

In addition, the target operating system also has an effect on the potential user base; many people run Microsoft operating systems while few use, say, OS/2, so the number of eligible users for Windows applications will be higher than for OS/2.

The target operating system has been found to be a useful component in predicting the success of an open source project [105].

3.2.2.10 Portability

Portability has the potential to affect the size of the target audience. The more platforms that a project is ported to, the more eligible developers and eligible users there are that may become involved with the project. Therefore, it initially appears that FLOSS projects with multiple ports are at a distinct advantage because of their ability to tap into a larger segment of the population. However, porting does not come without costs, as it introduces the complexity of maintaining code for multiple platforms. In an ideal scenario, most code can be shared across platforms, but in some cases there may be large bodies of platform-specific code that must be written and maintained for each supported system. The design of a project may even need to be significantly modified in order to allow convenient parallel development of ports (e.g., redesigning the software so that platform-specific code resides in separate modules). Side effects of the added complexities may mean less frequent releases of the software, an increased number of defects, more developers required, the necessity of developers to possess a wider range of skills in the form of platform-specific knowledge, etc.

One study [81] categorized projects according to the level of portability: single platform, hard-coded for multiple platforms, or multi-platform using an automated system. No significant difference was found between successful and unsuccessful projects based on the three portability categories [81].

3.2.2.11 Version Control and Software Configuration Management

Tools like CVS, SVN, and Git are popular with open source projects to provide both version control and software configuration management. SCM is considered a necessity for high maturity organizations to produce quality software as evidenced by its inclusion in CMMI level 2 [126]. SCM allows multiple developers to work on software in parallel and solves many problems pertaining to coordination. The ability for many programmers to simultaneously work on the same project allows for rapid development that is a positive and well-recognized characteristic of FLOSS development [34]. Furthermore, making SCM repository read access public means that everyone, including users, has access to the leading edge version of the code. This means that the latest bug fixes can be acquired as soon as they are available, rather than waiting until the next stable version is released, albeit with the risk that other more serious bugs may exist in development versions of a project. Making SCM repositories public also allows everyone to see where development is currently occurring and may make it more attractive to other developers by highlighting the areas of code that need work [81].

It has been found that the use of SCM tools is more common in successful projects than unsuccessful projects [81] and that most projects using SCM grant read access to the public [81]. However, it has also been found that projects that use SCM are no more efficient than projects that do not [75].

3.2.2.12 Mailing Lists and Forums

Mailing lists and forums serve as media for fast communication and may exist for multiple purposes. For example, developers use mailing lists and forums to discuss and coordinate work on a FLOSS project. Users employ lists and forums both to ask questions and provide

feedback on the project, and the content of these messages may influence future changes and improvements to the project's code, documentation, etc. Some projects may include multiple mailing lists and forums, e.g., a separate developers' and users' list. Projects that include mailing lists and forums appear to have a competitive edge over those that do not because they are better able to utilize the feedback from the users to enhance the project. In addition, these projects are also able to jump start Linus' Law: by notifying developers of unsolved problems via lists or forums, the chances improve that someone will produce a good solution quickly.

Mailing lists archives serve to lighten the load on developers and users by reducing the time and energy expended addressing already answered questions. Furthermore, archives also make it easier for users to access help without facing the potentially intimidating task of posting a question and risking appearing inept. If a question has already been answered, searching the archives is likely a faster path to obtaining an answer than posting an inquiry and waiting for a reply. Mailing lists and forums themselves are a mechanism to contribute to a project, and the help provided in these communication media may lead to a larger number of people involved in the project.

It has been found that successful FLOSS projects make better use of mailing lists [81] and more frequently include mailing list archives, with 80% of successful projects using archives compared to 50% of unsuccessful projects [81]. However, the use of mailing lists and/or forums has not been shown to increase the efficiency of an open source project [75].

3.2.2.13 Development Stage

The development stage of a project provides information about the maturity of the software. As such, it may impact the developers and users that choose to be involved with the software, thus influencing the success of the project.

Users likely are interested in a project for its current utility and thus may be drawn to projects in more mature stages. A rule of thumb is that projects listed in beta or later stages are ready for general use while projects in pre-beta stages lack significant functionality and are less likely to be refined to a level that would be considered user friendly.

The motivation for developers to be involved in a FLOSS project varies over a wide range. At the purest level, a developer may be interested in scratching his/her own personal itch [13], in which case the development stage may be less important than how close the project is to actually solving the developer's problem. Some developers may be motivated by other factors, such as maximizing their reputation in the FLOSS community [1], [13], [35], [98], [119], [127], in which case projects in earlier development stages may be preferable, as there are still opportunities to contribute to long-lasting core code [45]. Other developers may be looking to increase their future employment prospects by gaining skills and experience working on FLOSS [1], in which case development stage is less important than the potential learning opportunities that a project offers. There remain other reasons that developers report as motivation to be involved in open source projects that may or may not revolve around development stage. It therefore remains unclear how development stage affects the attraction of developers to a project.

Depending on the dynamics, development stage may be an indicator of a self-exciting process with a tipping point. That is, if a project requires a minimum number of developers and users to self-sustain, once that number is reached the project may accel-

erate towards success, picking up more users and developers as it also advances through development stages. Unfortunately these dynamics, and the role that development stage plays in them, are not well understood.

Software development is a dynamic process with different activities occurring depending on the development stage. [71] cautions that the expectations and goals of a project vary over the course of development and therefore developers may evaluate the perceived subjective and objective performance of a project differently depending on its development stage. For example, it is suggested that accomplishing clearly defined goals is a more important evaluation criterion for projects in later development stages. During the earlier stages clear working procedures and routines have yet to be established and therefore goal-oriented evaluation may not be appropriate [71]. However, developers accumulate project management skills over time, meaning improved performance may be expected as a project progresses [128]. On the other hand, this means joining a project in the later stages requires more effort on the part of the developer to get up to speed on the design [129] and processes. Keeping this in mind, it may be a project's development stage in combination with other factors that influence a developer's choice to join and stay with a particular project.

Development stage has been shown to be useful in predicting a project's success [105].

3.2.2.14 Activity Level

The progress of a project is tied to the level of activity. Activity level might also be considered a measure of the intensity of cooperation [66], with cooperation being an important and necessary component for creating public goods such as FLOSS. High levels of activity might therefore lead to high levels of cooperation and subsequently a project's success.

Projects with high activity levels have indeed been shown to be in more advanced development stages [66].

Using machine learning, [48] identifies antecedents that can be used to predict the success of projects at an early stage of development. Principle Component Analysis is used to determine the most important factors. During the first nine months of development, the number of distinct email posters, bug reporters, bug fixers, and SCM committers were found to be interchangeable factors that accounted for 69% of the total variance when predicting project success. Since these counts act as proxies for level of activity, this demonstrates that activity level is useful for predicting the success of a project.

Both of the aforementioned studies seem to support the simple notion that highly active projects are successful. What is less clear is what is the cause versus the effect. Do high activity levels lead to continued or increasing activity levels, perhaps by drawing attention to a project that then gains additional users and developers? Or are there other properties of projects that attract users and developers, and high activity levels are simply a side effect of a project improving and moving towards success?

3.2.2.15 Number of Developers

Underlying many of the previous sections is a simple notion: a project must attract at least a minimal number of developers in order to progress and self-sustain. Having developers associated with a project does not guarantee its success, but it certainly improves the odds, whereas a project with zero developers has no chance of becoming successful. As pointed out in Section 3.2.2.14, activity is necessary for a project to progress, and active projects appear to be more successful. It follows that the number of developers affects the level of activity and consequently might influence a project's success.

[66] tested the hypothesis that FLOSS projects with more developers are more successful in that they are in more mature development stages, are more active, and are more frequently used. The findings were mixed and only weakly supported the link between the number of developers and success [66]. However, [48] found that the number of committers could be used to predict the success of a project. The committers of a project are often those developers belonging to the core team, so this supports the concept that the number of core developers influences the success of a project.

3.3 Developer Motivation

Via a large horizontal study of approximately 400 FLOSS projects, [74] concludes that the pool of developers is limited and the resources available from developers is also scarce, especially with respect to the number of FLOSS projects competing for developers and their proficiencies. [64] similarly points out that making meaningful contributions to open source software involves skills, and because there is a limited pool of individuals with the necessary knowledge, experience, and expertise, projects may compete to attract the efforts of these talented developers [64]. As a result, in order to understand which projects will succeed it becomes important to gain insight into how developers select projects. The idea is that if developers are attracted to a project, the project will progress. However, if a project cannot both attract and retain developers, the project will remain inert and, by most success metrics, be considered (at least temporarily) dead [74], [77].

The purpose of this research is not to understand what initially motivates individuals to become involved with FLOSS. There exists a body of literature that already addresses why people contribute to open source or cooperative communities in general. Motivations that have been suggested include:

- To gain reputation in the hacker community.
- To increase employment opportunities by both learning from expert programmers and showcasing one's own talent.
- To create software that fulfills a personal need.
- To increase the quality of one's own work through the peer review that the FLOSS community provides.
- To learn new skills.
- To share knowledge and skills, to be altruistic, and to engage in the gift-giving culture.
- To have fun, because developers enjoy programming and find the experience of working on FLOSS satisfying.
- To realize a software project that could not be written without the help of others.
- Belief that software should not be a proprietary good.
- To limit the power of certain commercial software companies.

Several large-scale surveys (e.g., [1], [98]) have been conducted that specifically ask developers why they participate in FLOSS. There are also a number of arguments for developer motivation based on economic, social, and psychological theories (e.g., [13], [119], [127], [130], [131]). Overviews of motivational factors are provided in [15], [35], [70].

Instead of concentrating on the reasons contributors are attracted to FLOSS in the first place, this research focuses on how contributors choose which projects to join once they have already decided to be involved in the open source community. The development of FLOSS is a cooperative effort driven by volunteers, and therefore attracting and retaining

volunteers is imperative for a project to progress and, potentially, succeed [70]. [64] argues that in order for a project to be successful, it must attract contributions from developers, and that developers make their selections based on properties of the projects. Therefore, in order to better understand the FLOSS development process, this research attempts to identify the characteristics of a project that are important to attract developers.

To gain an understanding of what project characteristics cause individuals to choose one project over another, the following five factors are selected that are believed to influence individuals' decisions when picking a project: similarity between an individual and a project, current resources being contributed to a project, cumulative resources of a project, number of downloads a project has received, and development stage or maturity of the project. These factors are incorporated into the model via a utility function (see (6.2) on page 156) and then the model is used to explore the currently-unknown importance of each factor. The five factors were chosen based on existing literature covering influential factors and a general understanding of the FLOSS development process. The specific reasons why each factor is considered important to individuals selecting projects are explained in the following subsections.

3.3.1 Similarity

How important is the similarity between a project and an individual? That is, how closely must the aspects of a project match the interests of the individual in order to warrant the individual becoming and/or staying involved with the project? For example, an individual who has just acquired a new printer is more likely to be interested in FLOSS printing system projects than in peer-to-peer file sharing applications, at least in the immediate future.

FLOSS development is a voluntary process; developers both volunteer to be involved with a FLOSS project and then self-assign themselves to specific FLOSS tasks within the project [42]. This is a key difference from proprietary software engineering, where developers are assigned by management to projects and tasks that they might find boring, tedious, or even be unqualified for, whereas in FLOSS developers are free to choose projects and tasks that interest them[34]. There is little incentive for a developer to contribute to uninteresting projects, and a bored FLOSS developer may simply move on to other more interesting tasks, projects, or, if nothing appeals, leave the open source community altogether. Thus, the similarity between the interest of an individual and the characteristics of a project may be an important factor when selecting projects.

Similarity may also play a role in the enjoyment that developers experience from working on an open source project. Many FLOSS developers claim to enjoy the work they are doing [130]. In fact some developers use their level of personal enjoyment working on a project as metric for judging if the project is successful [133]. Other motivations for

---

[34]The self-assignment of FLOSS developers to tasks may be seen as a key advantage of the FLOSS development process. Indeed, while discussing practices of high maturity organizations, an anonymous participant at a Software Engineering Institute (SEI) workshop for CMM level 4 and 5 organizations observed that '"Getting the right person into the right job on the project is still the most important aspect of project success. People are not plug-compatible. The expertise of individuals is critical. Process is an enabler; not a replacement."' [132]. By allowing developers to choose tasks that interest them, FLOSS significantly increases the chances of a well-qualified person working on each task. This concept is affirmed by Linus' law, which states that "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone." [13]. In other words, through the inherent transparency and openness of the process, FLOSS frequently manages to have the right person fix the right problem. Furthermore, FLOSS developers may be motivated to complete their tasks at an above average level simply because they are interested, and therefore motivated, in the tasks they have chosen.

participating also seem to point to similarity being important. For example, a survey found that 44.9% of developers participate in FLOSS development for intellectual stimulation [98] and a subgroup of FLOSS developers may be considered "fun seekers" [98]. Presumably, similarity plays a role in determining if a project is fun and/or intellectually stimulating.

Finally, software is typically created from a developer's personal need, i.e., the desire for a developer to scratch his/her own personal itch [13], and solving a software problem is often listed as a reason for being involved in a FLOSS project [1], [13], [52], [131]. Similarity may therefore be important because individuals are interested in projects that address problems that are similar to their own.

3.3.2 Current Resources

The current resources being contributed to a project is an indicator of project activity level. In addition to providing the literal value measuring the amount of work being completed, it also might be used as a proxy to indicate how many active developers are currently contributing to a project – that is, it indicates the popularity of a project with active developers. This is because generally a project with more active developers will have more resources being contributed.

Research shows that active projects are desireable. One study [67] considered freshmeat's vitality score, which is an indicator of developer effort, and freshmeat's popularity score, which is calculated based on the number of people subscribed to a project and the number of hits to the project's homepage. It was found that an increase in a project's vitality led to an increase in popularity as well.

[113] argues that membership herding occurs in FLOSS, meaning individuals joining or leaving a project encourages other individuals to also join or leave the project re-

spectively. Indeed, a survey of 34 developers showed that 76% agreed that membership dynamics were critical to projects progressing continuously [113]. Furthermore, developers indicated that being involved in an active community was energizing and important to keep a project progressing [113]. Lack of activity on a project was cited as a compelling reason to leave and potentially find a different project to contribute to [113]. It has been shown that large scale-free network are prone to herding dynamics [113], and that the SourceForge project and developer network is indeed scale-free [57].

It may be the case that popularity begets more popularity when it comes to open source projects. It has been argued that through the structure of the underlying social network, the number of members associated with a project may affect the attractiveness of the project to other potential members [57]. [127] states that open access projects, such as FLOSS, are stigmergic because they include a positive feedback cycle that encourages more work to be completed in the future based on the work that is currently being performed.

In terms of consumers, active projects may be perceived as offering better support, improved chances of adding needed features in the near future, etc. On the other hand, too much activity may be an indicator of an unstable projects. Users may grow tired of the need to frequently upgrade the software to fix bugs. Likewise, users may become frustrated if each frequent upgrade introduces a plethora of new bugs or changes components, such as the user interface, that the user must then relearn. This is more likely to occur with projects that do not include stable releases but instead encourage users and developers alike to download the code from the development branch[35]. Thus while project activity may

[35] An exception to this, where stable versions have been released frequently, might be the web browser Chromium, http://www.chromium.org, which released an incredible six major

be important to consumers as well, there may be a happy medium between stagnant and excessively active that consumers find most desireable.

3.3.3 Cumulative Resources

Cumulative resources is a measure of the size of a project completed so far. Developers may take into consideration the size of a project in concert with other factors, such as the number of developers involved, when selecting a project. For example, a small project will need fewer developers than a large project. Consequently, it may be easier for a developer to find tasks to complete on a large project than to break into a small group of developers already engaged in a small project.

In addition, developers may be interested in different size projects depending on the time commitments they are able to make. Some may be interested in short-term projects or projects that require a minimum investment in order to get up to speed. In general, programmers find it easier to write new code than to understand and work on someone else's code, yet the majority of a developer's time is spent on the latter task. This is likely true when joining an existing project that already has a code base; the existing source code, project design, etc. must first be understood before contributions enhancing the software can be made. Therefore, a small project may allow a developer to get up to speed faster because there is less code to understand before he/she can start making meaningful contributions. On the other hand, a developer seeking a project he/she can work on long-term may not mind the initial investment in time.

---

versions in 11 months, four of which occurred in five months; in one case, version 9 was released just 15 days after the initial release of version 8 [134].

As projects progress, knowledge among the developers accumulates [128]. For developers who are motivated to pick up new skills, this may be an incentive to join projects that already have a large code base; projects with minimal or no code may not have enough work completed that can be learned from, and therefore may be of less interest to developers looking to improve their skills.

Consumers, likewise, may consider the size when selecting a project. For example, a user looking for an email client might be interested in something minimalistic (e.g. Alpine, a text-based email client derived from the freeware-licensed Pine) or feature rich (e.g., Evolution, a GUI-based email client that includes additional components such as a calendar, address book, task organizer, etc.). The size of a project therefore helps a user determine how lean or bloated the software is.

3.3.4 Download Count

The number of times a project has been downloaded may be used as a proxy for the importance and popularity of a project with users.

[57] simplifies existing literature [135] on classifying participants of FLOSS development. [118] recognizes two course-grained classification groups of users and developers, further splitting users into passive and active and developers into peripheral, central, core, and project leader categories. Passive users are defined as those who utilize the software without contributing anything to the respective project while active users submit bug and feature requests and may be active in forums or mailing lists [57]. When analyzing developer communities for the purpose of modeling FLOSS social networks, [57] chooses to ignore passive users because they do not contribute directly to FLOSS projects and therefore [57] does not consider them developers. However, passive users, also known as free-

riders, do influence a project; namely, by using a project, passive users are acknowledging the utility of the project and value of the work of the developer team. If reputation and acknowledgment of one's skills motivates developers to engage in open source development, then passive users must be included in the model. Indeed, it has been shown that increased user interest in a project results in increased development activity [64].

[136] analyzes crowdsourcing behavior through data from YouTube[36]. Although creating videos is slightly different than developing software, the general concepts are the same. In the case of YouTube, the content being developed is videos instead of software, but in both instances the content is created by volunteers through collaboration and published for consumption by the masses. [136] finds that the number of videos produced by a user is strongly correlated to the number of times a user's previous videos have been downloaded. A lack of video downloads often leads to users uploading fewer videos, asymptotically approaching zero. This is similar to a developer abandoning a project because no one is using the software. [136] suggests that the digital commons can be regarded as a private good where the participants are paid in recognition for their efforts by download tallies instead of money. This is in alignment with research that has shown that people are sometimes willing to skip financial gain for attention[37] [137]. Recognition has also been shown to be important in some online communities [138]. Thus the number of users of a specific project may indeed influence a developer when selecting software.

---

[36]http://www.YouTube.com

[37]An example of this in academia is publishing papers and monitoring citations [136]. The authors of a paper receive recognition for their contributions every time someone cites their paper, when a respected journal agrees to publish their work, etc. While there is no direct monetary compensation, the recognition and attention gained is often sufficient incentive to continue writing papers.

In FLOSSSim, passive users affect a download count. Each time a user downloads a project, either the initial time the user tries the software or when updating to a newer version, the download count of the project is incremented. In this way, the download count becomes a measure of the popularity of a project with users. Thus developers may gauge the importance of a project by how large the user base is and use this as a proxy for how many people will benefit from contributions to the project[38]. Note that in FLOSS, developers are more often than not also users of the software they develop. FLOSSSim includes developer's downloads in the download count.

3.3.5 Maturity

The maturity property of a project is unique from the other factors considered in that it is a human-assigned value. Rather than creating a set of rules or manually judging projects' maturity based on a set of criteria, instead folksonomy principles are used to categorize projects by using the development status. By using the developer-assigned development stage, values are used which were assigned by experts who have critically evaluated the projects – namely, the administrators of the projects themselves. The criteria used by each developer may differ, but the law of averages should reduce any major biases introduced because of this; because this is a subjective measurement, a high level of variance is both expected and considered acceptable.

---

[38]This is similar to some open source projects which allow users to vote on which bugs should be fixed first. In other words, the users explicitly try to channel developers' efforts to the problems the users find most significant. In the same way, users are "voting" by using a project, and developers trolling for a project to work on may consider the number of users as an indicator of the importance of the project.

The maturity of a project influences both the objective and subjective performance of a project [71]; specifically, the subjective assessment of a project by developers may depend on the development stage [71]. Furthermore, research shows that the development stage plays a significant role in predicting the success, and more specifically the level of activity, of an open source project [105]. Therefore, it makes sense to explore the importance of development stages to developers who are selecting projects.

If developers are interested in increasing reputation, [45] argues that developers will prefer creating core rather than fringe code because the core code is likely to be included in many releases of the project, thus giving developers a long-term boost to their reputation. In this case, developers will prefer projects in lower development stages because there remains core code to be written. However, very early development stages, which may possess the greatest potential for reputation gain, also include significant risk that the project will die before producing useful software, in which case there will be no reputation gain for the developers involved.

At the other extreme, when a project becomes fully functional and enters the production/stable and mature stages, it may also become less desirable to developers. First, if reputation is a driving force, by this time most of the high-profile tasks are complete [67]. Secondly, the majority of the functionality is already implemented and the project transitions from development to maintenance, meaning most of the "personal itches" that cause developers to join a project have already been scratched [67].

According to [13], in order for open source project development to commence, there must exist at least a kernel of working code to begin with. Bazaar-style development[39] can be used for testing, debugging, and improving software once a program base is established, but it is almost impossible to start a project in this mode. In other words, it is advantageous for an initial version of the software to be developed cathedral-style (often this is just a single programmer hashing out a solution to his/her own problem) before releasing the software to the open source community. For this reason, projects that are made available to the open source community before there is an initial working version, such as projects posted in the planning or pre-alpha stages, may be at a serious disadvantage when it comes to attracting developers [67].

To test if projects released into open source at later development stages are more likely to reach higher development stages, data was mined from the FLOSSmole database[40]. The development stage of projects on SourceForge in April 2009 was recorded, along with the development stage when each project first appeared on SourceForge. The number of projects in each development stage as of April 2009 versus projects' initial stages when added to SourceForge is shown in Table 3.2. Of mature projects, only 1.7% started in the lowest two stages (planning and pre-alpha) while almost three times as many (5.1%) started in the alpha, beta, or production/stable stage. Similarly, for projects currently in the

---

[39] [13] famously coined the terms "cathedral" and "bazaar" to refer to traditional, closed source development and open source development respectively. Like meticulously constructing a cathedral, proprietary software is carefully built by talented people working in isolation. Open source, on the other hand, resembles a great babbling bazaar, with many differing ideas and approaches mashed together, yet out of the chaos materializes stable software.

[40] See Section 4.1.3.2 for a description of the FLOSSmole database.

TABLE 3.2
Number of projects in each development stage as of April 2009 versus projects' development stages when first added to SourceForge.

		April 2009 Stage											
		planning		pre-alpha		alpha		beta		stable		mature	
Initial Stage	planning	17894	(99.39%)	445	(3.63%)	316	(2.39%)	357	(2.14%)	221	(1.81%)	7	(0.76%)
	pre-alpha	53	(0.29%)	11713	(95.66%)	500	(3.78%)	452	(2.71%)	210	(1.72%)	9	(0.98%)
	alpha	23	(0.13%)	41	(0.33%)	12323	(93.24%)	872	(5.23%)	436	(3.57%)	5	(0.54%)
	beta	21	(0.12%)	35	(0.29%)	63	(0.48%)	14923	(89.55%)	1012	(8.28%)	16	(1.74%)
	stable	13	(0.07%)	7	(0.06%)	13	(0.10%)	55	(0.33%)	10330	(84.57%)	26	(2.82%)
	mature	0	(0.00%)	3	(0.02%)	1	(0.01%)	5	(0.03%)	6	(0.05%)	859	(93.17%)
	Total	18004		12244		13216		16664		12215		922	

production/stable phase, only 3.5% started in planning or pre-alpha while more than three times as many (11.9%) started in the alpha or beta stages. This supports the notion that development stage is relevant to the success of a project since projects released as open source in later stages are more likely to increase in development stage.

Additionally, as can be seen by looking at the main diagonal, Table 3.2 also highlights that most projects never change from the development stage they started in. It is difficult to know if this phenomenon is real or a side effect that could be produced simply by project developers failing to update a project's status (developers are forced to supply a development stage when registering a project on SourceForge but there is no mechanism to force and/or remind developers to update this status indicator as the project progresses). Table 3.2 also shows that some projects go backwards, ending in an earlier stage than they started in. In some cases this can be expected, such as when a stable project creates a new major release and moves back to the beta stage. In other cases, such as the 13 projects that started in the production/stable stage and then moved back to planning, it is likely an error in the data.

3.4 CONCLUSION

The goal of this research is to better understand the FLOSS development process, with a particular interest in understanding why some projects are successful while others are not.

To understand what it means for a FLOSS project to be successful, success must first be defined. Unlike proprietary software, there is no universally accepted method for determining if a FLOSS project is successful. Since traditional metrics do not necessarily apply, a number of FLOSS-specific metrics have been proposed. Many of these metrics have not been well-studied or applied to real data. As such, the meanings and impact of using different success metrics is further explored using FLOSSSim in Section 7.1.4.1.

Understanding why some projects are successful and others are not involves examining factors that may influence the success of a project. Analyzing the impact of factors ranges from examining technical attributes of existing projects to borrowing concepts from public goods theory. Fortunately, much research has already been performed on understanding factors that may be antecedents to success, and a comprehensive review of these factors and findings has been provided.

This research focuses not on what motivates individuals to become involved with FLOSS in the first place but rather on how individuals select projects from the large pool of available FLOSS. Five components believed to be important to individuals choosing projects are selected. Through the use of agent-based modeling, the actual importance of each of these factors will be explored.

Finally, there is currently minimal research on FLOSS consumers. To further this largely unexplored segment of open source, this research includes modeling consumers for the purpose of better understanding the users' influence on the FLOSS development process.

CHAPTER 4

DATA

In order to calibrate and validate the model, data about the FLOSS development process must be obtained. As pointed out by [139], traditional research in social science is theory-driven. It involves well-designed surveys, information relatively free of errors, and tends to be small-scale. However, with the Internet comes the ability to collect huge amounts of data about social activities, including information on the development of FLOSS, that are available due to the online nature of these activities. From this is born a new, non-traditional approach to social science research that is data-driven. Often it involves crawling large-scale networks to extract the necessary data from noisy environments [139]. Both theory- and data-driven methods are used in this research, although there is more focus on the data-driven approach due to the nature of the available data.

The following sections provide information about the types of data and data sources that are available, and outline some of the problems associated with the data.

4.1 DATA SOURCES

A common problem when creating models of social systems is a general lack of data for design and validation purposes. Fortunately, the Internet is a domain where data pertaining to social phenomena is often available. This is because online activities, from creating Facebook pages to editing Wikipedia articles to updating blogs, typically leave a digital trail. In some cases, these data may be readily available; in other cases, it may be necessary to

perform data mining activities (sometimes spread out over time, such as looking at updates to a web page over a year) in order to extract the required information.

FLOSS development falls into the online activity category. In general, there are several mechanisms for obtaining FLOSS data, including:

- Surveys and literature
- FLOSS hosting sites
- Databases
- Extraction tools

Each of these categories is further covered in the following sections.

4.1.1 Surveys and Literature

FLOSS is actively being researched and there are many papers published on FLOSS. Although the motivations of the research varies widely, many of the findings for FLOSS research not directly related to predicting the success of FLOSS projects can still be applied to modeling open source software development.

Both technical and social data are available from existing literature. In the case of technical data, this often means researchers have collected and aggregated data using the other techniques described in Sections 4.1.2, 4.1.3, and 4.1.4. Examples of useful data in this category include distributions, such as what percentage of projects have one developer, two developers, etc. [117] or what percentage of developers are working on one project, two projects, etc. [1].

Social data often come in the form of surveys and interviews. There are several major and well-known surveys in the FLOSS field (e.g., [1], [98]) and many other smaller

surveys as well. These surveys cover a number of different components of FLOSS, including, but not limited to:

- Developer demographics (e.g., age, geographic locations, marital status, occupation) [1], [98], [130], [131], [140]
- Developer motivations (e.g., why developers contribute to FLOSS, how developers choose projects, and what causes developers to stop contributing) [98], [130], [131], [140]
- Developer contributions (e.g., average time spent working on FLOSS, number of LOC contributed) [98], [130], [140]
- FLOSS usage (e.g., percent market penetration of certain FLOSS) [93], [141]

Of the categories enumerated for obtaining FLOSS data, surveys and interviews are the most direct mechanism for gathering social data from the developers themselves. The remaining categories focus more on the technical data, although it may be possible to gain insight into social behaviors by studying, for example, trends in technical data as well.

4.1.2 FLOSS Hosting Sites

By its very nature, FLOSS development tends to be distributed, with volunteers potentially involved from multiple geographic locations. The Internet has broken down geographic boundaries and allows people to interact with others around the world with a minimum investment via communication mechanisms such as email. In the case of FLOSS development, there exist a number of FLOSS hosting sites that facilitate open source development. These sites typically provide tools and resources to help with open source development including, but not limited to:

- Project hosting repository
- Software configuration management/version control tools
- Bug organizing and tracking software
- Feature organizing and tracking software
- Membership controls (e.g., control over who can commit code)
- Email lists and archives

Sites that provide collections of tools and services for collaborative software development are called "forges" [142]. Forges are not inherently limited to FLOSS and may also be employed for closed source software development.

At time of writing, the largest and most famous FLOSS site is SourceForge[1], hosting 306,464 projects[2] [144] and continuing to grow. The SourceForge site tracks projects, with publicly accessible project data granularity ranging from hourly (for the last 48 hours) to monthly (for data older than 30 days), depending on the data [145]. Select data, namely data that does not violate privacy policies, from the SourceForge site is available for academic research [146]. SourceForge started operation in November 1999, meaning over a decade of projects' history has been recorded by the site.

---

[1] http://SourceForge.net

[2] There are multiple methods for retrieving the number of projects hosted at SourceForge and the numbers reported differ. The number quoted here is based on the count provided in SourceForge's search. According to SourceForge's sitemap, there are 448,114 projects [143]. It is difficult to compare the two counts to discover the discrepancies, but it is likely that the larger list contains all projects that have ever been registered at SourceForge while the smaller list is missing projects that have been removed from the site.

Interestingly, SourceForge is itself comprised of open source software [147], [148] that is, at least partially, hosted at SourceForge[3].

Other FLOSS hosting sites include:

- Savannah (http://savannah.gnu.org and http://savannah.nongnu.org)
- BerliOS (http://www.berlios.de)
- Gna! (http://gna.org)
- Google Code (http://code.google.com)
- RubyForge (http://RubyForge.org)
- OW2 Forge (http://forge.ow2.org)
- Java Forge (http://javaforge.com)
- Tigris.org (http://www.tigris.org)

By their very nature, the tools at these sites track the progress of projects. Analyzing this data can provide insight into the FLOSS development process. Some examples of data that may be available include:

- Software Configuration Management (SCM) logs: Encompassing revision control, SCM tracks changes to the code, documentation, and other files associated with a project. For each change that is made, known as a "commit", the committer's identification, files affected, date and time, etc. are logged. Popular SCM software includes Concurrent Versions System (CVS), Subversion (SVN), and Git.

---

[3]The SourceForge project homepage is http://sourceforge.net/projects/sourceforge/

- Project ranks: A list of projects ordered by level of activity bounded by time. An activity score is typically calculated using multiple metrics from a project. A project rank may be used to compare projects hosted on the same site.

- Bug trackers: Bug tracking software provides a mechanism for organizing, prioritizing, and tracking bugs. Useful information, such as how quickly certain types of bugs are resolved, can be calculated using data from bug tracking software.

- Feature requests: The equivalent to bug tracking software, only for tracking enhancements and new features for a project.

- Forums: Email lists, wikis, chat logs, etc. provide historical data on discussions pertaining to the software being developed. Information that can be extracted from forums include how decisions are made, who is actively participating in the project, etc. Forums are a source of information for social data.

- Web statistics: This includes statistics that could be extracted from a web server log, such as the number of visitors to a project's homepage, the number of times a project has been downloaded, etc.

Note that not all hosting sites provide data to the public. In theory, however, this data exists, even if it is kept private by the hosting site. For those sites that do make some of the information available, it is typically accessible via web pages on a project-by-project basis. Manually probing this data may be sufficient when only a small number of projects are being studied; to collect data for a large number of projects, it may be necessary to use spidering and screen scraping techniques. As an example, data that is accessable by browsing SourceForge's website is contained in Table 4.1.

TABLE 4.1
Per project data tracked by SourceForge.

Data	Description
Registration date	The date a project was first registered at SourceForge.
License	The open source license used by a project.
Programming language	The programming language(s).
Operating system	Operating system(s) under which a project will run.
Topic	Selected from a set of predefined values such as networking, database, games/entertainment, telephony, editors, emulators, etc.
Translations	Original language and translations supported by the software.
Intended audience	Selected from a set of predefined values including end users/desktop, developers, manufacturing, government, etc.
User interface type	Selected from a set of predefined values including command-line, Win32, X Window System, web-based, Curses/Ncurses, etc.
Development stage	Selected from the following subset of values: planning, pre-alpha, alpha, beta, production/stable, and mature.
Release date	Date of the most recent release version.
Number of developers	The number of developers registered with the project.
Number of administrators	The number of developers registered as administrators with the project.
SCM	The number of reads and writes to the software repository.
Mailing list	Archive of messages posted to the mailing list.
Forum	Archive of posts to the forum.
Download count	Tracked individually for each project file.
Tracker	Includes the number of open and closed bug reports, support requests, feature requests, and patches.
Project web traffic	Divided into three components: the total number of files served (known as “hits”); the total number of times a project’s web logo is served (known as “pages”); and bandwidth.
Activity rank	An aggregate measurement that indicates how active a project is with respect to other projects. See Section 4.2.2.1 for more details on how activity rank is defined.
Popularity rank	The popularity of a project compared to all other projects.

4.1.3 Databases

For those researching FLOSS, obtaining data is a common task. Fortunately, there exist several FLOSS databases that, in the spirit of open source, have opened their data for use by others. In some cases these databases are provided by researchers who also needed data on FLOSS and, rather than forcing all researcher to duplicate their efforts, have made the collected data available to the public.

Pre-built databases provide a number of advantages over extracting the data from hosting sites. First, the tasks of crawling and screen scraping are avoided. Both of these tasks can be difficult, tedious, error prone, and time consuming. For example, crawling a forge may result in the IP address of the crawler being blocked by the site[4]. Including a delay between page requests reduces the chances of being blocked but also increases crawl time. Screen scrapers are sensitive to changes in web pages; even a small change to a web page may require the scraper to be modified. [82] outlines some of the difficulties specifically with spidering and screen scraping SourceForge. Thus by using existing databases, one can focus more on the data and less on obtaining the data.

In some cases, existing databases provide historical data that may not be available directly from a FLOSS hosting site. Consider the case where a hosting site only provides current, but not historical, data. If a database is built from multiple crawls of the site spread over time, the database becomes an archive of historical data. This same data cannot be extracted in a single pass from the site itself as only the current, and not the past values, are available.

---

[4]FLOSS researchers at Notre Dame tried to crawl SourceForge and caused the entire campus to be blocked from the site [149]!

Finally, placing the data in a database creates the ability to perform sophisticated queries using database query languages. These same queries would be much more difficult and error prone without the use of a database.

The following sections describe three existing FLOSS databases:

4.1.3.1 SourceForge Research Data Archive

Historically, SourceForge's policy has been to make a subset of the site's data available for research purposes [146]. Prior to 2006, obtaining this data meant submitting a support request to the SourceForge staff [150]. In 2006, a research project at the University of Notre Dame, aimed at understanding open source software[5], resulted in the Department of Computer Science and Engineering at Notre Dame agreeing to host data dumps from SourceForge [151]. The result is the SourceForge Research Data Archive[6] (SRDA). Each month, SourceForge creates a snapshot of the site's backend database, cleans the data of private and sensitive information, and then makes it available to Notre Dame [152], [153]. The SRDA is a collection of these snapshots.

Requests for access to the SRDA can be gained by completing a questionnaire and signing a sublicense agreement [154]. Access is granted on a case-by-case basis [155], and the data is only available to academic researchers[7] [154], [155].

---

[5]http://www.nd.edu/~oss/

[6]http://srda.cse.nd.edu/

[7]Prior to the creation of the SRDA, SourceForge's policy did not limit research data to be used for academic purposes only [150].

Access is available only via the web or a web service [153]. The SourceForge contract prohibits Notre Dame from providing dumps to researchers for local processing [154].

4.1.3.2 FLOSSmole

FLOSSmole[8], formerly OSSmole, is an open source project with the purpose of obtaining and making publicly available data about open source projects [156]. The project maintains a set of tools for spidering forges and converting the collected data into multiple formats [156]. Crawls of forges are performed regularly and the data are made available to the public. In addition, the software tools employed in the collecting and parsing process are available under an open source license so that researchers can use them to collect their own data [156]. Finally, the project accepts data donations, which are then integrated into the the existing data sets and also made available for public use. Several well-known American FLOSS researchers are involved with this project.

To increase the quality of FLOSSmole's data, the HTML of the crawled pages is also stored in the database [156]. This affords the possibility of reparsing the data at a later date should a change in the HTML cause incorrect parsing but not be noticed during the initial processing of the data. In addition, this increases versatility by allowing additional data not originally extracted from the crawled pages to be added to the database at a later time.

Unlike the SRDA, FLOSSmole follows the spirit of an open source and makes its data available to anyone, researchers or otherwise [156]. Both academic and non-academic

---

[8] http://flossmole.org/

users are welcome to the data and in most cases the data can be retrieved and used without any form of registration. Data is available in three formats [157]:

1) Flat files: These are text files containing parsed information.

2) SQL files: These files contain "CREATE" and "INSERT" statements and can be used with the open source MySQL database[9] to create a local copy of the database.

3) Direct database access: This is an existing online MySQL database populated with all the FLOSSmole data that interested users can request remote access to.

In addition to SourceForge, FLOSSmole also collects and/or maintains donated data from the following sites [157]:

- freshmeat (http://www.freshmeat.net)
- RubyForge (http://rubyforge.org)
- OW2 Forge (http://forge.objectweb.org)
- Free Software Foundation (http://directory.fsf.org)
- SourceKibitzer (defunct) (http://www.sourcekibitzer.org)
- Savannah (http://savannah.gnu.org)
- GitHub (http://github.com)

FLOSSmole is itself partially hosted at SourceForge and, more recently, partially hosted at Google Code.

---

[9]http://www.mysql.com/

The first crawls were performed in 2004 [157]. Different forges have been added over the years so data dating back to 2004 is not available for all forges. The intended collection interval is 2 months but complications sometimes cause larger gaps in the data. For example, a major site redesign of SourceForge in July 2009 broke FLOSSmole's tools and caused regularly-scheduled crawls to be missed. In addition, forges differ in the data that is available; therefore FLOSSmole captures different data from different forges.

4.1.3.3 FLOSSMetrics

FLOSSMetrics[10] is the newest of the three pre-built databases, commencing work in September 2006 and scheduled to last 36 months [158], [159]. Funded by the European Commission, and including the involvement of many European leaders in FLOSS research, this project aims to create and analyze a large-scale public database using existing data, proven techniques, and already available software tools [158], [159]. Providing data for the calibration of FLOSS simulation models is included in the list of reasons for creating FLOSSMetrics [158].

FLOSSMetrics collects data from four types of data sources [159]:

1) SCM repositories

2) Source code

3) Mailing list archives

4) Bug tracking systems

---

[10]http://flossmetrics.org/

Data is collected in an automated fashion via a set of software tools [159]. These tools, many of them existing prior to the start of the FLOSSMetrics project, locate FLOSS repository data, parse SCM logs, analyze source code, parse mailing lists, store information in database tables, etc. The tools are open source themselves and are available from http://forge.morfeo-project.org/projects/libresoft-tools/ .

FLOSSMetric's data is available to the general public via http://melquiades.flossmetrics.org . Database files can be downloaded on a per project basis or as an aggregated file covering all projects [160]. An API to access the data is also under development [160]. Data is divided into three categories: SCM and code metrics, mailing list information, and bug tracking information [160]. Data for all three categories may not be available for all projects [160], [161]. Although FLOSSMetrics started collecting data more recently than FLOSSmole and the SRDA, historical data is included since the data inputs (e.g., SCM logs, mailing lists, and bug tracking logs) inherently contain historical data. This is a distinct advantage over crawling methods used by FLOSSmole, which relies on regular collections to create historical data.

FLOSSMetrics continues to increase the number of projects included in the project's data. An initial run of 1000 projects was available in March 2009 [159]. As of December 2009 this had expanded to include at least partial coverage of approximately 2800 projects [161], [162]. A goal of the FLOSSMetrics project is to provide data for a minimum of 5000 projects [158], [159].

4.1.4 Extraction Tools

A final method for obtaining data on FLOSS projects is to use extraction tools. A collection of tools, for example, is used to build the FLOSSMetrics database. However, not all projects

are included in the FLOSSMetrics database and therefore the tools may be used to locally supplement the existing data with projects of interest currently missing from the data set. There may also be additional project information that is available but not currently extracted by any of the pre-built databases. In these cases, it may be necessary to use and/or modify an existing extraction tool in order to obtain the desired data.

The following list provides a short sample of the data extraction tools that are available:

- OSSmole Tools: Tools to spider and retrieve project information from SourceForge, currently used by FLOSSmole [163], [164].
- CVSAnalY: Tool to parse SCM logs and store extracted data in a database [165]. Data extracted include the date, files being modified, type of modification, developer making the commit, etc. [165]. CVSAnalY supports CVS, SVN, and Git repositories [165].
- Mailing List Stats: Tool to analyze mbox format email files and store extracted data in a database [166]. Data extracted includes email addresses, email fields (e.g., to, from, date, subject fields), message body, mailing list info, etc. [166].
- Bicho: Tool to analyze bug tracking software logs and store the extracted data in a database [167]. Data extracted includes bug id, description, priority, status, comments, etc. [167].

For full details on the exact data collected by each of these tools, please see the software's associated documentation.

4.1.5 Data Sources Used

All of the above described data sources were considered for this research, but due to a number of factors (e.g., availability of desired data, ease of retrieval, data format, data licenses), only the following subset were used in this research: existing surveys and literature, manual retrieval from SourceForge site, FLOSSmole and FLOSSMetrics databases, and CVSAnalY extraction tool.

4.2 Data Caveats

With the large number of data sources collecting copious amounts of data, FLOSS initially appears to be ideally suited for modeling, having sufficient data for calibration and validation purposes. Unfortunately, a closer examination of the data available reveals there are considerable problems, and what initially appears as ideal data may, in some cases, be unusable.

The following sections provide a brief overview of generic problems with online data, followed by problems specific to the FLOSS data used in this research.

4.2.1 Problems with Online Data

Most data sets contain errors, creating what is known as "dirty" data. No matter how carefully data is acquired, virtually all large data sets contain at least some errors [168], [169], [170]. These errors may be introduced by humans, bad sensors, etc. For example, a FLOSS developer may enter incorrect metadata about a project. Some errors result in data inconsistencies that may be detectable by creating impossible combinations or contradictory information (e.g., having a null value where null is not an option, or having two people with the same social security number) known as data integrity violations.

Combining data from different sources may be necessary in order to obtain a sufficiently large or unbiased data set. Data collected from different sources may be heterogeneous, causing complications when combining the data into a single database. There also may be differences in data schemas among data sources that must be resolved using techniques such as transformations [171]. Even if the schemas are identical across sources, there may still be differences in the conventions used to collect the data at each source [172], yielding data inconsistencies that may be difficult to detect. Duplicate data is also a potential problem when combining data from multiple sources. Detecting duplicate data may be non-trivial, such as when time stamps for data vary slightly but all other fields are identical [172], or when errors already exist in the data. The detection and elimination of duplicate data is commonly referred to as the merge/purge problem [173].

Missing data is another common complication. Cases where a field is populated for some records and missing for others can yield distorted, misleading, or even useless query results. Removing records with missing data may result in a biased data set. Null values may be introduced when combining data from heterogeneous sources or performing imperfect transformations, in addition to the situation where data was never entered at the source.

Data is not always available in structured formats. Extracting data from unstructured or semi-structured sources increases the chances of creating dirty data. For example, screen scrapers may attempt to extract data fields from HTML. A small change in a web page may cause a scraper to miss certain fields or to collect the wrong information entirely [174].

Data cleansing is a field that includes techniques to detect errors in the data, detect missing data, assess the usability of the data, etc. [168], [171]. In some cases it may be

possible to fix dirty data that is detected [168]; in other instances, dirty data may simply be removed [171].

Although all data sets are expected to contain errors, when the number of errors is small with respect to the total number of data points, statistical methods, such as the law of averages, may reduce the effects of these errors.

In order to extract meaningful data from any data sources, it is necessary to first gain an understanding of the data, to investigate for potential problems and artifacts, and in general to critically evaluate the data. Errors will still be overlooked, especially for large data sets, but a careful prescreening of the data will help reduce erroneous and misleading query results (i.e., garbage in, garbage out). Having domain knowledge about the data may help detect problems.

4.2.2 Problems with FLOSS Data

A brief examination of the problems encountered while processing FLOSS data is provided in the following subsections.

4.2.2.1 Historical Data

Examining historical data provides information about the progress of FLOSS projects. For example, historical data can be used to examine the growth of a project over time. This means relying on data that is collected, potentially, over years. The temporal dimension of historical data introduces potential data complications of which to be wary.

The first concern is that data collection spread over time is consistent. Not surprisingly, sites such as SourceForge have changed over the years in order to better serve the community. This means making improvements to the system (e.g., enhancing the user interface), adding new tools as they become available, and removing obsoleted features. These

changes have both direct and indirect effects on the data that is collected. For example, SourceForge maintains an activity ranking of all projects. This is an aggregate metric that attempts to provide information indicating how "active" a project is, taking into account three components: project traffic (e.g., downloads), development (e.g., age of most recent release), and communication (e.g., forum posts) [84], [150]. In February 2005, SourceForge replaced the formula for calculating activity with a new formula that the SourceForge team felt better reflected the concept of activity [150], [175]. As a result, project activity rankings before and after this date cannot be directly compared. One must be careful if using this data to only sample before or after February 2005, or to perform transformations so the activity ranks are the same.

As another example, this one of tool changes that affect historical data, SourceForge originally offered CVS as the only option for an SCM tool. SVN was later developed to address some of the shortcomings of CVS [176] and around 2006, SourceForge added SVN as an option. Existing projects now had the possibility of migrating from CVS to SVN; projects could also migrate back to CVS if they found they preferred it. Likewise, for the first time new projects were faced with the choice of selecting a repository management system (also with the possibility of changing systems at a later date). This resulted in some projects using CVS, some using SVN, and some using both (presumably so project administrators still had access to the old versions of the project even while using the new tool). In addition to creating heterogeneous data that made it difficult to compare SCM statistics for projects using different tools (not to mention heterogeneous data for single projects that switched between CVS and SVN), this also introduced two problems that tainted historical data:

1) When migrating to a different tool, the entire software project is often added to the new system via a single commit. This distorts the SCM data as it appears that most of the code was written at a single time, by a single developer, etc.

2) Some projects migrated to a new tool and then chose to turn off the old tool. In this case, there was no longer any record of the development that had occurred using the old tool, further distorting the SCM data.

SourceForge has since added other SCM software, including Git [177], Mercurial [178], and Bazaar [179], exacerbating the problems associated with using SCM data.

SourceForge has also implemented several GUI makeovers/site redesigns to the SourceForge website. In addition to changing the look and feel of the site, some of the project data was relocated. For example, prior to a July 2009 redesign, all project metadata was available on a project's SourceForge homepage. After the redesign, some of the metadata, such as a project's development status and the number of developers working on a project, was relocated so it was included only in search results but no longer listed on a project's summary page. In addition to relocating these data, the data became more obscure by requiring a search option, which by default is off, to be turned on in order to view the information. Project metadata is important to the research presented here and after the site redesign it appeared it was no longer available on the SourceForge site. I was about to submit a bug report when I discovered another frustrated site user had found the data and was requesting it be relocated to the project summary page[11] [180]. When significant changes such as these are made, not just to the look and feel but also to the contents of the site,

[11] SourceForge closed the bug report, declaring this was actually a feature request, not a bug. As of August 25, 2011 project metadata is still not listed on a project's homepage.

it affects the historical data. In this case the metadata (or lack of metadata) changes how people locate, evaluate, and choose projects on the SourceForge website. As with the other changes affecting historical data, there is likely a difference in data logged before and after a major SourceForge site redesign and thus researcher must be wary when using data that spans these dates.

SourceForge's site redesign also changed how project administrators update metadata about their projects. The ease with which this data can be updated affects the data's accuracy, which in turn affects historical data from before and after the changes were made.

There are additional problems that are unique to collecting data via crawling. For example, even small site redesigns can break spidering and parsing software, resulting in holes in historical data. Although FLOSSmole's goal is to collect data from SourceForge every two months [181], the major site redesign in July 2009 has resulted in a much larger gap while the FLOSSmole team scrambled to update the spiders and parsers [163], [182]. Depending on the frequency of site changes, as well as the crawling frequency, this may or may not affect research using the data. FLOSSmole developers have expressed great frustration about the constant need to rewrite software to adjust to the changes made by SourceForge [182], [183]; there is even talk of abandoning SourceForge crawls entirely, since there are other sources for this data, and focusing instead on crawling the sites for which FLOSSmole collects data exclusively [182], [183]. To decrease the impact of changes and increase the collected data quality, FLOSSmole stores the spidered HTML files in the database. By doing this, parsers can be changed and rerun at a later date to correct data extraction problems. Unfortunately, changes to the structure of a website may cause a spider to not crawl and store the correct pages, an error which cannot be corrected at a later time.

Spidering is an inherently expensive task; crawling activities frequently include built-in delays to prevent overloading a server and being denied access to the data via denial of service attack protection mechanisms. FLOSSmole has, for example, not collected certain available data, such as data not accessible without visiting another set of web pages, simply because the cost of visiting additional web pages is too high [183]. In addition, not all components of large sites like SourceForge may be fully functional or even available during a crawl. In these cases, parts of the crawl may be missed due to partial site outages (see [184] for an example of where this has occurred in a FLOSSmole crawl). Direct database dumps, such as SRDA, are not subject to this problem.

A final problem with crawled data is it may not include data from the inception of the repository. For example, over the years FLOSSmole has expanded the number of sites that are crawled. Therefore, the data archives for some sites may not include very old data, even though the particular forge may be older. A direct data dump of a forge, on the other hand, does include data from day one of the repository's operation.

4.2.2.2 Cleansing Data

The goal of data cleansing is to both 1) detect dirty data and 2) fix errors whenever possible. Unfortunately, the best that can sometimes be done is to simply discard the dirty data and retain the clean data. As long as only a small percentage of the data are dirty, this may be an acceptable data cleansing method. Lamentably, some of the data encountered during this research included fields that were frequently populated with erroneous values. Having no way to fix the data, these records were discarded; what remained was sometimes too small a data set to be useful. See Section 6.2.2.1 for examples of data filtering that resulted in substantially smaller subsets of the starting data set.

With sufficiently large data sets, it is possible to minimize the effect of (potentially undetected) errors using statistical methods (e.g., the law of averages). These methods are effective in cases where the amount of dirty data is small compared to the size of the data set. Inherently, because of the sheer amount of data collected on FLOSS, one expects these methods to be effective. Unfortunately, this research found much of the data to be highly contaminated and therefore statistical techniques may not be particularly effective.

Finally, care must be taken when filtering the data to avoid introducing unwanted side effects or biases. For example, the majority of FLOSS projects are unsuccessful and dormant; these projects produce little interesting data. Consider examining trends in the number of developers working on a project. The majority of projects will have a single developer for their entire duration. Filtering out one developer projects may yield more interesting trends, but it also eliminates a huge number of FLOSS projects. It may have eliminated the majority of unsuccessful projects (along with a few successful projects). Thus, the data set after filtering is likely biased towards successful projects. Therefore, one must be careful when performing any form of filtering during the data cleansing process to avoid unintentionally introducing undesirable side effects. See [82] for a discussion on how filtering may affect correlations.

4.2.2.3 Missing and Misleading Data

Regardless of the data source, it is important to carefully examine and understand the original data in order to avoid a "garbage in, garbage out" scenario. In particular, there are a number of caveats in FLOSS data that lead to missing or, worse yet, misleading data. Some examples of these are provided in this section.

In the case of crawling and parsing to obtain data, there is a problem of variation in the pages being crawled. Crawling a single source, such as SourceForge, where the avail-

able data and layout are standardized, decreases the chances of incorrectly collecting data. Unfortunately, even sites that use back-end databases and HTML templates to generate their displayable content contain exceptions and special cases that can lead to missed or incorrect data being collected. For example, unexpectedly long fields or special characters can break regular expression searches. [82] mentions difficulties differentiating HTML code from user names that start with "<" and end with ">" (e.g. "<foo>"). Misspellings and variances in case (e.g., "foo.bar@asu.edu" versus "foo.bar@ASU.EDU") can introduce bugs in the data (e.g., treating a single entity as two separate entities). Bad links, such as human entered URL's, can also cause problems with spidering. Without manually checking all data, it is hard to know how many exceptions occurred in a crawling and parsing run. Discovering even a small number of errors can make one suspicious of the validity of the data [82]. In cases where pre-built databases are used, such as FLOSSmole or FLOSSMetrics, it may be wise to read the documentation and mailing lists to understand how the crawls were performed and what problems others have already encountered in the data. Pre-built databases offer the advantage that, like with FLOSS development, many eyes have already inspected the data, discovered the problems, and addressed some of the issues, resulting in a cleaner data set.

Crawlers are also sensitive to site outages. FLOSSmole, for example, has encountered trouble retrieving the 60 day statistics for projects, which are frequently unavailable [184]. While the FLOSSmole spider retries multiple times to retrieve these statistics, it eventually gives up, leaving a hole in the data collected [184]. If possible, it is recommended to check a site's status before commencing a crawl [82], although due to the size of some sites and the amount of time it takes to completely traverse all links, it may not be possible to complete the crawl without encountering at least some site outages.

Some FLOSS projects choose to create a "placeholder" account on a well-known forge but then host the actual project elsewhere. Presumably, because SourceForge has become the largest open source hosting site, developers like to keep a pointer on SourceForge so people searching for FLOSS will still find the project. Hosting the project elsewhere may come as a personal preference, such as for the flexibility to use whatever tools are preferred by the developers rather than the limited choices provided by SourceForge. Unfortunately, this semi-common practice leads to distorted data. For example, the XFree86 Project, an open source implementation of the X Windows System, has a project page on SourceForge[12] but maintains the real project homepage, CVS repository, etc. elsewhere[13]. The data on SourceForge for this project is inaccurate and outdated; the project appears inactive with version 4.6.0 being release in May 2006 [185]. The external project homepage includes version 4.8.0, released in December 2008, with evidence of further development occurring in 2009 [186] and support still being offered in 2011 [186]. Thus, the data on SourceForge is misleading and regardless of how the data is obtained (i.e., via a manual visit, an automated crawl, or a direct dump of SourceForge data) will misrepresent the project. Compounding the problem is that outside hosting seems to occur more frequently with successful projects. This means that data for successful projects, a small subgroup of the projects hosted at SourceForge to begin with, is also the most likely to be incorrect due to hosting offsite. It is difficult to detect projects in this category without human inspection of the SourceForge project pages.

---

[12]http://sourceforge.net/projects/xfree86/

[13]http://www.xfree86.org

The opposite problem also occurs: some projects are founded and developed elsewhere and then are migrated to SourceForge. This also produces misleading data, such as the project start date, which will reflect when the project was registered at SourceForge, not when development actually commenced. An example of this is Tcl[14] [82].

Certain metrics that are commonly applied when measuring FLOSS also have inherent complications. For example, the number of times a project has been downloaded is tempting to use as a proxy for the popularity, or even the success, of a project. The download count, unfortunately, only shows part of a project's usage. SourceForge tracks download counts but only for officially released files. Users are also often granted anonymous read access to the SCM repositories. This means users can checkout the current developer version of the code at any time – without affecting the official download count (although this will increment a "read" counter in CVS or SVN). For projects that are rapidly developing, using the latest version may be preferable, as it includes the latest features and bug fixes. Some projects strive to keep their developer version stable through nightly builds/tests and encourage users to checkout the latest version rather than official release versions. Since code checkouts are not included in the download count, the download count will underrepresent the actual number of times the software was downloaded. Summing the download and SCM read counts also does not provide a good popularity measure; project developers, for example, will frequently checkout and modify code, thus inflating the SCM count even though the popularity, in terms of number of users, is not increasing.

---

[14]http://sourceforge.net/projects/tcl/

Some projects are very popular but infrequently downloaded, which further distorts using a project's download count as an indicator [82]. For example, projects like vim[15] (a vi-like text editor) are included in many UNIX distributions. Users of UNIX systems use vim frequently, but a system administrator almost never downloads and installs a new version of vim. Thus, the download count for certain projects performs as a poor indicator for measuring popularity.

The number of developers involved with a project is another commonly used metric that also is problematic. Most projects have a small set of core developers that engage in the majority of development [51], [77], [78], [79]. However, most FLOSS projects also accept code contributions from anyone. These fringe contributors typically do not have write access to a project; rather, peripheral developers submit patches to the core developers, who then commit the changes to the repository [187]. If simply analyzing the number of developers via straightforward methods (e.g., on SourceForge, looking at the list of developers registered to work on a project; in general, analyzing an SCM log and counting the unique committers), this number will be smaller than the actual number of contributors since there will be no record of peripheral developers. A more accurate count may be obtained by manually combing through the source code, which likely includes the names of those who contributed patches (attribution is a requirement of many open source license), and examining email lists, where contributions may appear in the form of code snippets, pseudocode, algorithm descriptions, etc. In addition, there are non-traditional developers that won't appear in SCM logs. This includes people who have contributed documentation or taken the time to accurately describe bugs and workarounds, etc. While not traditional

[15]http://vim.sourceforge.net/

code contributions, these are crucial in the FLOSS development process to help produce high quality, usable software.

While much data about FLOSS developers is available, there is almost no data available about users of FLOSS. This is partially due to the difficulty of collecting user data. Unlike developers, who must register with a project in order to directly contribute, users are able to download and use FLOSS projects anonymously. Automated data collection aside, it is also difficult to target users with surveys, again because the users are largely anonymous. Regardless, since users are also stakeholders in FLOSS projects, user data may be beneficial in better understanding the whole FLOSS process.

Finally, while using existing studies provides a quick and easy mechanism for obtaining data, care must be taken in choosing which data are applicable. Modeling FLOSS requires an understanding of data across many projects, and as already pointed out, collecting data across large numbers of projects can be a difficult task. For this reason, there are many case studies that look at a single or a handful of FLOSS projects, e.g., [51] looks at the Apache Web Server, [54], [130], [188] study Linux, [123] analyzes FreeBSD, [79], [189] consider GNOME, and [187] uses Python as a case study for the research performed. Obviously Apache, Linux, FreeBSD, GNOME, and Python are not the norm; these are wildly successful projects and therefore are likely different than the average FLOSS project. This is in fact a problem with most case studies, which almost always consider successful, well-known projects. Indeed, individual case studies have already shown differences between successful projects [42] and therefore relying on studies of single or just a few projects may provide non-generalizable results [42].

4.2.2.4 Integrating Data

As outlined in Section 4.1.3, there are multiple sources of FLOSS data. Unfortunately, the data is heterogeneous and therefore combining across multiple data sources is non-trivial. Some of the pre-built databases attempt to address this problem: FLOSSMetrics, for example, collects project data from multiple sites but utilizes tools to homogenize the data. An example of this is FLOSSMetrics' use of CVSAnalY to collect the common data (e.g., committer, date of commit) from multiple SCM systems [165].

The added complications of integrating data from multiple sources make using a single data source an attractive option. However, using a single source may not be a reasonable option for the following reasons:

1) Limits the number of projects: In many cases, a single data source may not have a sufficient number of projects to study. This is one reason why SourceForge is popular with researchers; the site is sufficiently large and thus using data only from this site eliminates the complications of using multiple smaller sites and combining the data in order to obtain a sufficiently large group of projects to study.

2) May introduce a bias: Certain sites cater to certain types of projects. For example, RubyForge specializes in FLOSS projects written in Ruby. Savannah is the home of GNU projects, introducing a bias towards the GPL and LGPL licenses. Including multiple sources helps to eliminate (potentially unintentional) biases introduced by poor sampling techniques.

In some cases, even data that should align perfectly does not. A number of differences have been discovered between FLOSSmetrics and the SRDA, even though both of

these data sets represent the same data source. Understanding why there are discrepancies can be a time consuming process.

4.3 CONCLUSION

An attractive aspect of modeling FLOSS, as compared to other social systems, is the abundance of data available due to the online nature of open source development. There are multiple approaches for gathering this data that range from using existing databases intended for research to manually retrieving the data oneself. As has been illustrated, care must be taken with all the data used, as what initially appears to be clean is often data contaminated with artifacts, errors, missing components, etc. Therefore, it is necessary to carefully explore and understand the data before using it. In some cases, it may be necessary to clean the data by performing transformations, pruning, etc. Care must be taken not to add unintentional biases when going through this process.

CHAPTER 5

FLOSSSIMPLE

The progress of digital public goods such as FLOSS relies on projects being able to attract volunteers. One of the assumptions is that the success of a project affects the attractiveness of a project. However, a key problem in the literature of open source software is the ambiguity of the definition of success. Can it be measured by the number of downloads, the frequency of releases, the number of bug fixes, or any number of other indicators (see Section 3.1.2 for a more comprehensive list of possible metrics)?

In this chapter, a very simple, theoretical model of the evolution of populations of digital public goods is presented; the model, called FLOSSSimple, is used to explore the consequences of different definitions of success on FLOSS. After exploring success in this simple model, a more complex and comprehensive model of the FLOSS development process is presented in Chapter 6.

First, basic empirical findings from studies on open source software are presented in Section 5.1. A description of FLOSSSimple is provided in Section 5.2 followed by an analysis of the model in Section 5.3.

5.1 Characteristics of FLOSS Contributions

It has been shown that contributions to open source are highly skewed. For example, the distribution of developers working on a project is highly distorted, with 67% of projects having only one developer and 90% of projects having fewer than 4 developers [117]. It has also been found that 10% of the developers write 72.3% of the code [15], and the

100 most active developers on SourceForge are involved in an astounding 1886 different projects [80].

In order to comprehend why the contributions to FLOSS are so skewed, it is necessary to understand why people contribute to or use certain digital public goods. A possible explanation is that contributors prefer to participate in successful projects. However, what it means for an open source project to be considered successful is ill-defined (see Section 3.1.2 for a list of proposed FLOSS success metrics). As a result, various success metrics that reflect the accumulation of activities and the number of individuals involved are applied to the model and used to explore whether different definitions of success have an impact on the patterns generated by the model.

The model presented here is based on public goods theory. Although the theory is specifically applied to creating a model of FLOSS, the model may likely be adapted to explore other digital public goods with minimal changes.

5.2 Model Description

The model presented is a very simplistic model of consumers and producers of an ecology of public goods based on the observed processes of open source software development. Given are N_{a} agents and N_{p} projects. At each time step an agent may 1) contribute to the development of a project and/or 2) consume (a.k.a. use) a project. Each agent has a probability p_{c} to contribute to a project and a probability p_{u} to use a project. The probabilities p_{c} and p_{u} are drawn from an exponential distribution with $\lambda = 0.1$ and truncated to values in the range $[0.0, 1.0]$. This represents the notion that most agents will have a small probability to be active during a time step.

When an agent is active during a time step it will make a decision about which project to contribute to or use based on how close the characteristics of a project match with

the preferences of the agent. To define how well agent preferences match characteristics of a project, the matching interests value M_{i} is calculated for each project, as shown in (5.1):

$$M_{\mathrm{i}} = 1 - (n_{\mathrm{p}} - n_{\mathrm{a}})^2 \tag{5.1}$$

where n_{p} is the characteristics (a.k.a. needs) of the project and n_{a} the preference value for the agent[1]. If both dimensions match, M_{i} is equal to 1. Initially values for n_{p} are assigned randomly from a uniform distribution. However, it is assumed that agents biased towards consuming may have certain needs (e.g., an interest in well-documented, easy-to-use projects) while agents biased towards producing will have a different set of needs (e.g., an interest in projects for the challenge and to gain experience) [1], [98]. Thus values for n_{a} are based on an agent's producer number p_{c} and consumer number p_{u}. A simple function for mapping from p_{c} and p_{u} to n_{a} is shown in (5.2):

$$n_{\mathrm{a}} = \mathrm{f}(p_{\mathrm{u}}, p_{\mathrm{c}}) = \frac{p_{\mathrm{c}} - p_{\mathrm{u}} + 1}{2} \tag{5.2}$$

A visual depiction of the mapping is shown in Fig. 5.1. Essentially, an agent's needs can be thought of as a continuum between 0.0 and 1.0. Lower values ($n_{\mathrm{a}} < 0.5$) represent a bias towards consuming and higher values ($n_{\mathrm{a}} > 0.5$) a bias towards producing. For a given point (x,y), mapping is done by finding the nearest point on the agent needs line and then projecting this onto the second y-axis. Thus a full-strength producing agent has a needs vector of 1 ($\mathrm{f}(0,1) = 1$), a full strength consuming agent a needs vector of 0 ($\mathrm{f}(1,0) = 0$),

---

[1]To keep variable names consistent between FLOSSSimple and the more complex model FLOSSSim presented in Chapter 6, n_{p} and n_{a} are referred to as needs vectors. However, in this simplified model these are one-dimensional vectors and can thus be treated as scalars.

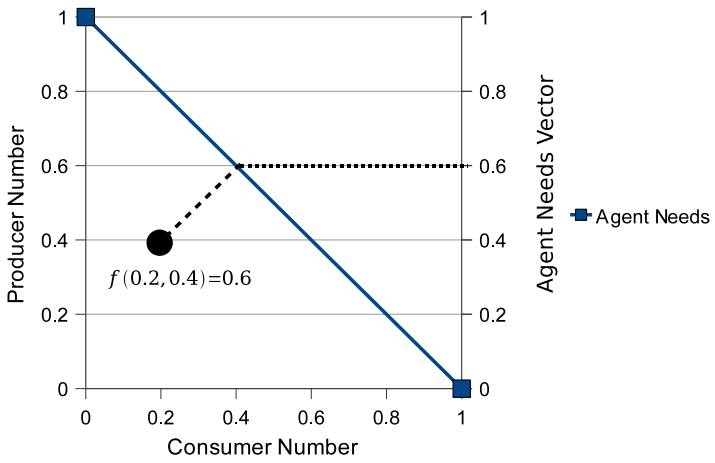


Fig. 5.1. Example of mapping an agent's producer and consumer numbers to its needs vector. For a given point (x,y), the nearest point on the agent needs line is found and then projected onto the right y-axis.

and an equal-part producing, equal-part consuming agent a needs vector of 0.5 ($\mathrm{f}(x,x) = 0.5$).

The utility U_{u} of a project to a consuming agent is proportional to how well the agent's and project's interests match, as shown in (5.3):

$$U_{\mathrm{u}} = M_{\mathrm{i}} \tag{5.3}$$

Agents who decide to contribute to a project are assumed to take into account both how well the project matches the agent's interests (M_{i}) and how successful the project is (S_i). Agents differ in their weights, α, for the two different indicators. Currently, α is drawn from a normal distribution with a mean of $1/2$ and standard deviation of $1/6$. The utility of a project for contributing agents is thus initially defined as

$$U'_{\mathrm{c}} = \alpha \cdot M_{\mathrm{i}} + (1-\alpha) \cdot S_i \tag{5.4}$$

where M_{i} and S_i are normalized to values between 0 and 1. Since there is no agreement on how to measure success S_i, the consequences of the following different formulations of success are explored:

- S_1: current number of consumers
- S_2: cumulative number of consumptions
- S_3: current number of producers
- S_4: cumulative number of contributions (a.k.a. work completed)
- S_5: each project equally successful

S_1 and S_3 represent the current popularity, or recent popularity, of a project with consumers and producers respectively. S_2 and S_4 represent long-term popularity with consumers and producers respectively. S_5 is used as a baseline for comparison purposes. Note that each success metric is normalized. For example, for a given project P, S_1 is the current number of consumers working on P divided by the total number of agents currently consuming any existing projects.

Finally, a switching cost sc is subtracted from the utility of production if the project is different than the last project the agent contributed to. This reflects the transaction costs in switching projects, which may include learning new customs and becoming familiar with code. The final utility for contributing agents is defined in (5.5):

$$U_{\text{c}} = \alpha \cdot M_{\text{i}} + (1-\alpha) \cdot S_i - sc \tag{5.5}$$

If an agent makes a decision to consume or produce, it calculates the expected utility of all available projects and chooses the project with the highest utility.

When a project is not updated (no agent contributes to it) for a number of time steps (using a default of 5 time steps), it is considered a dead or inactive project and is removed from the system. Agents who produce can start a new project with a probability $p_s \cdot p_c$, where p_s is a model-level constant controlling the probability of creating a new project. Thus agents who have a higher tendency p_c to contribute to projects are also more eager to start new projects. In the default case p_s is equal to 0.01.

Agents only contribute to a project when they expect this will lead to a utility greater than or equal to a minimum utility $U_{\min}$. Agents stay with a project they last worked on if they choose to participate and there is no better alternative. In the default case $U_{\min}$ is equal to 0.2.

Agents who contribute to a project will affect the characteristics of the project n_p. The new value of n_p at time t (i.e., $n_{p,t}$) is adjusted by the average values of the preferences of the contributing agents' n_a values and the previous value of n_p (i.e., $n_{p,t-1}$). The adjustment rate between a project's old characteristics $n_{p,t-1}$ and the contributing agents' preferences n_a is determined by λ, which is equal to 0.85 in the default case, as shown in (5.6):

$$n_{p,t} = \lambda \cdot n_{p,t-1} + \frac{1-\lambda}{totcont} \cdot \sum_{i=1}^{totcont} n_{a,i} \tag{5.6}$$

where $totcont$ is the current number of agents contributing to a project and $n_{a,i}$ is the preference of the ith agent contributing to the project.

Agents may not reconsume the same project within a certain time span, modeling the assumption that consuming an unchanged or minorly changed project will not be worth the effort (e.g., there is effort involved in downloading and installing software). The time span limiting reconsumption is set to 10 for the model runs considered.

The model is implemented in NetLogo[2]; multiple runs were executed in parallel on a high performance computing system at Arizona State University[3].

5.3 Model Analysis

To study the model, two input parameters are varied and several resulting distributions analyzed. The input variables considered are the definition of success and the switching cost. The five possible success metrics were outlined as S_1–S_5 in Section 5.2. The switching cost, *sc*, is a penalty subtracted from the utility of projects the agent did not work on during the last time step the agent contributed to a project, as described in (5.5). This represents the extra effort an agent must expend to familiarize itself with a new project before it can provide meaningful contributions. Values considered for switching costs are 0.0, 0.1, 0.2, and 0.4. Note that utility values range between 0.0 and 1.0 so a penalty of 0.4 is significant.

The model was run for 1000 time steps, thought to be sufficient to allow conditions to stabilize, and populated with 1000 agents. Each parameter combination was run 10 times and the results averaged to account for the stochastic nature of the model.

5.3.1 Cumulative Resources and Consumer and Producer Distributions

The following distributions were considered after the 1000^{th} time step:

- Cumulative resources (i.e., amount of work) a project has accumulated
- Number of consumers using a project
- Number of producers contributing to a project

---

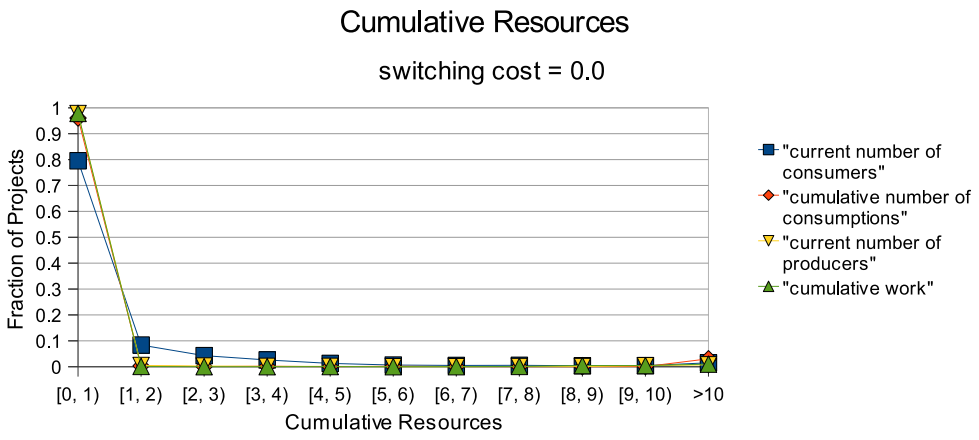
[2]http://ccl.northwestern.edu/netlogo/

[3]http://hpc.asu.edu

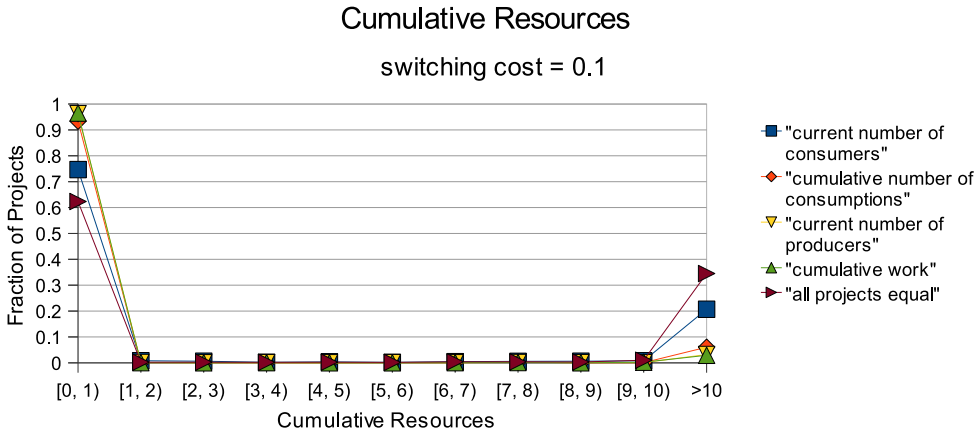
All results were normalized for comparison of the distributions. The results for switching costs 0.0 and 0.1, and for the five different success formulations for the three different output data are depicted in Figs. 5.2–5.4.

The *all projects equal* success metric is an outlier when the switching cost is 0.0. This is for the following reason: by causing all projects' success to be equal, the S term in the producer utility calculation U_c (5.5) becomes the same for all projects, causing utility (and thus selection) to be based entirely on an agent's and project's matching interests M_i. Essentially, treating all projects as equally successful is the equivalent to selecting a random project for S, the opposite end of the spectrum of using best choice. When paired with a switching cost of 0.0, there is a lack of stability in agents working long-term on certain projects. Instead, agents rapidly flit from one project to another, especially as new projects are created that better match their interests. This results in an explosion in the population of projects, since projects rarely exist for 5 time steps without being worked on by an agent and thus are rarely removed from the simulation. As a result, data from this success metric are not included in the figures when the switching cost is 0.0. For non-zero switching costs the *all projects equal* success metric is used as a baseline to show that agents discriminating among projects using other success metrics does have effect on the resulting distributions.

With a switching cost of 0.0, the cumulative resources distribution is similar for three out of four of the success metrics, with the *current number of consumers* being the exception, as shown in Fig. 5.2. Essentially, most projects accumulate zero or almost no resources. For all success metrics, as the switching cost increases, the cumulative resources morphs towards a bimodal distribution. Projects are either small or large, and the higher the switching cost the greater the number of large projects. A non-zero switching cost essentially creates a "stickiness" factor; agents tend to continue, or stick, with the project they



(a)



(b)

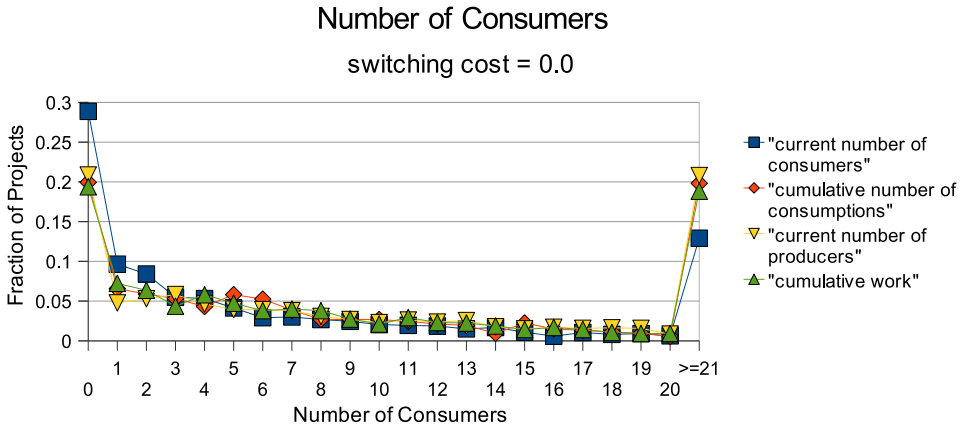
Fig. 5.2. Fraction of projects as a function of cumulative resources. (a) is with switching cost *sc* equal to 0.0 and (b) is with switching cost *sc* equal to 0.1.

were last involved in simply because the penalty of switching to a different project overwhelms the utility of doing so. Thus projects tend to be small, by never attracting agents, or large, by having agents stick with them for extended periods and thus accumulating a substantial amount of work. This is similar to data observed on SourceForge, where most projects never get off the ground, but a few accumulate enough code to become useful software. Note that with a switching cost of 0.1, two groupings emerge: the *cumulative number of consumptions*, *current number of producers*, and *cumulative work* distributions change

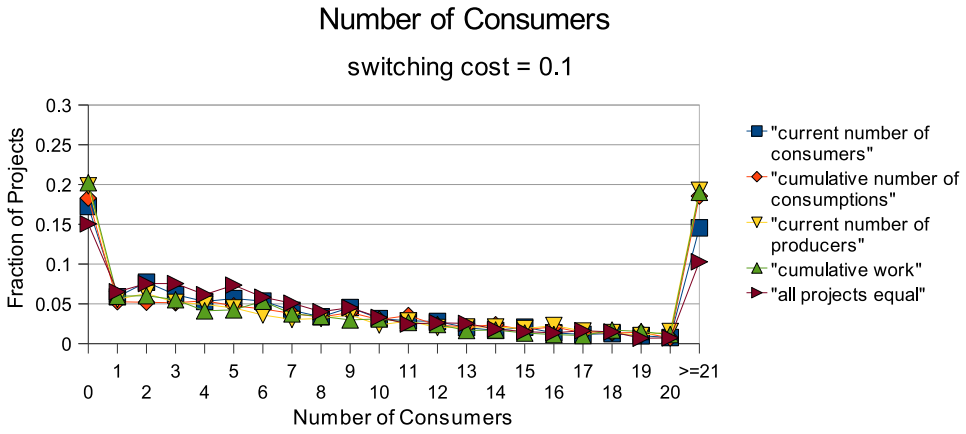
little with the increased switching cost, while *current number of consumers* becomes more of an outlier. In fact the *current number of consumers* distribution is very similar to the *all projects equal* distribution, indicating this metric is not very discriminating. This is because consumers are restricted from reconsuming the same project twice within 10 time steps. Thus, consuming agents are constantly moving through a subset of favored projects, which causes this metric to perform more like random selection. Note that producers behave at the other extreme: as the switching cost increases, a producing agent is more and more likely to contribute to the same project in the subsequent time step.

In general, most projects have very few consumers and few projects have many consumers, with a semi-smooth trend connecting the two extremes as shown in Fig. 5.3. Once again, the *current number of consumers* distribution is an outlier when the switching cost is 0.0. All other success metrics, regardless of the switching cost, seem to result in similar distributions, including the baseline of *all projects equal*. This is expected since, unlike contributing agents, consuming agents do not consider a project's success when calculating its utility U_{u}. Also unlike contributing agents, consuming agents do not include a switching cost in their utility function; therefore, large differences are not expected among the success metrics or by varying the switching cost. Indeed, the minor variations seen must be side effects of other components of the model, such as where producers are contributing.

The number of producers distributions are shown in Fig. 5.4. As was seen in the cumulative resources distributions, once again the *current number of consumers* metric is less similar to the other success metrics and more similar to the baseline of *all projects equal*. This supports the notion that the *current number of consumers* metric is non-discriminating. As the switching cost increases, more producers are associated with projects. This is similar to the results of the cumulative resources distribution and again can be explained by the



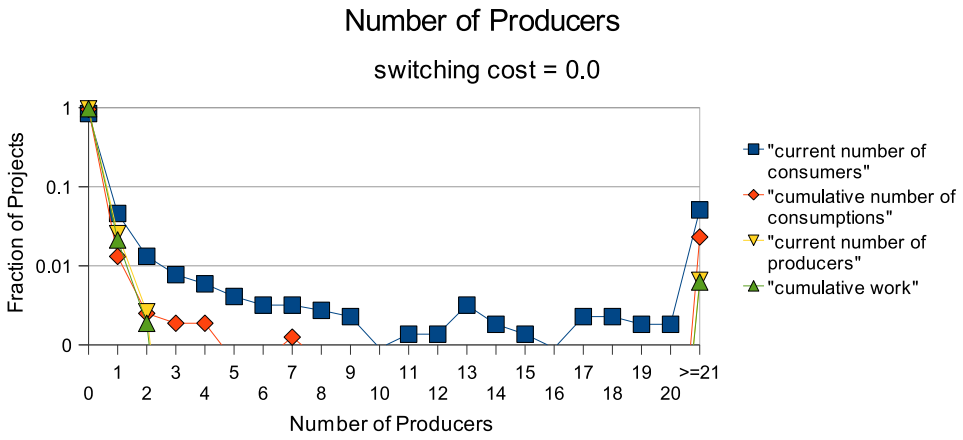
(a)



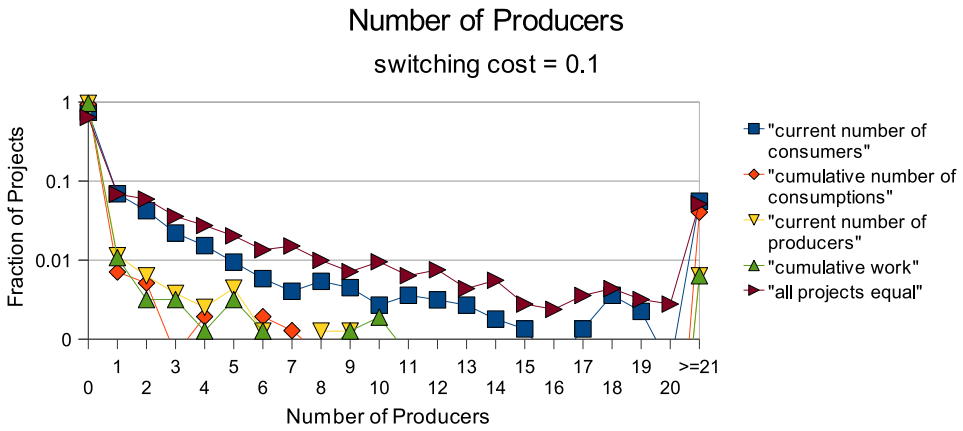
(b)

Fig. 5.3. Fraction of projects as a function of the number of consumers. (a) is with switching cost *sc* equal to 0.0 and (b) is with switching cost *sc* equal to 0.1.

increased stickiness factor. As the switching cost increases, ignored projects are removed, but those that do attract the attention of producers tend to keep those producers and gain additional producers, which also stay with the project long-term, over time. Note the highly skewed distribution from the model mimics data from studies of SourceForge. Averaged over all success metrics and with a switching cost of 0.1, 73% of the model's projects have zero or one developers and 85% have fewer than four developers, as compared to 67% and 90% respectively for SourceForge data [117].



(a)



(b)

Fig. 5.4. Fraction of projects as a function of the number of producers. (a) is with switching cost *sc* equal to 0.0 and (b) is with switching costs *sc* equal to 0.1.

At this level, the evaluated success metrics show only minor variations for the observed distributions. The one exception is the *current number of consumers* metric, which tracks closely with the baseline *all projects equal*, meaning it is similar to random choice. Since the success metrics only impact the producers, it is not surprising that more variation is seen when looking at cumulative resources and the number of producers distributions, since these distributions are directly linked to producers whereas the number of consumers distribution is only indirectly linked. Finally, switching cost also has an effect on the model,

with projects having more producers and accumulating more resources when switching costs are high.

5.3.2 Projects' Needs Vector Distributions

Part of the goal of this research is to better understand what types of projects are successful. Recall that new projects are continually added during a simulation run while projects that do not receive contributions for 5 time steps are removed. Thus, at any given time the population of projects consists of active projects, as inactive projects are continually being eliminated. By examining the projects' needs vector distribution, an understanding can be gained about what types of projects survive and are therefore arguably successful. Fig. 5.5 shows the distribution of projects' needs vectors for each of the five success metrics. There is not significant variation with different switching costs and therefore only the switching cost of 0.1 is shown. Note that the peaks of the distributions are slightly off center. Recall that needs vectors less than 0.5 are biased towards consumers and greater than 0.5 towards producers (see (5.2)). The distributions are skewed to the right, with peaks occurring in the range [0.5, 0.6), supporting the notion that surviving projects tend to be biased to producer needs. This is in alignment with literature that argues open source development is a producer-driven process (e.g., [13], [66], [101]).

Finally, some additional values of surviving projects are examined. Fig. 5.6 contains scatterplots of project needs vectors versus cumulative resources. Each plot contains the results for 10 runs of the model with a switching cost of 0.1. There are separate plots for each of the non-baseline success metrics. Consumer-oriented success metrics cause projects to accumulate a wide range of resources over a wide range of needs vectors, as shown in Figs. 5.6a and 5.6b. On the other hand, only a very narrow range of needs vectors accumu-

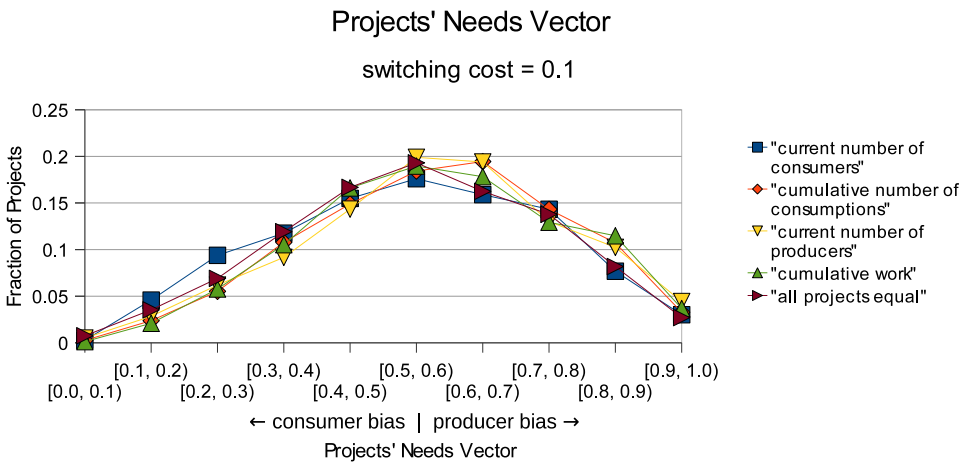
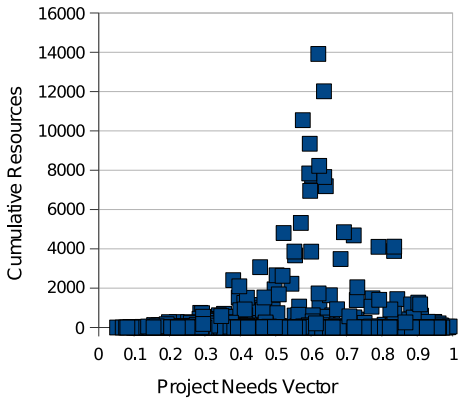


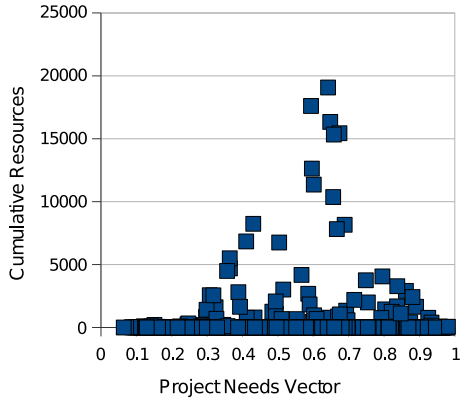
Fig. 5.5. Fraction of surviving projects as a function of projects' needs vectors. Values less than 0.5 indicate a consumer bias and greater than 0.5 indicate a producer bias.

late resources for producer-oriented success metrics, as shown in Figs. 5.6c and 5.6d. The behavior of the producer-oriented metrics is closer to that observed at SourceForge, where only a small percentage of projects develop beyond a few lines of code. Notice that for all success metrics the maximum resources accumulate for projects with needs vectors around 0.6, or those projects slightly biased towards producers.

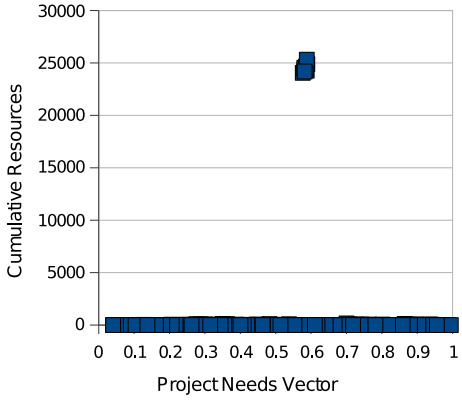
Fig. 5.7 shows scatterplots of project needs vectors versus cumulative consumptions. Notice that the peak consumptions occur at project needs vectors less than 0.5, showing a consumer bias. This is logical, as consumers, and not producers, affect the number of times a project is downloaded. Consumer-oriented metrics result in more consumed projects with low needs vectors. To understand why, recall that only producers affect which projects survive. When calculating the utility U_c of a project, producers consider two factors: the matching interests M_i and the success S. The matching interests is always biased towards producer-oriented projects. Thus when using producer-oriented success metrics, producing agents mostly select projects with needs vectors greater than 0.5. On the other hand, using consumer-oriented success metrics causes the second term in the utility function to favor



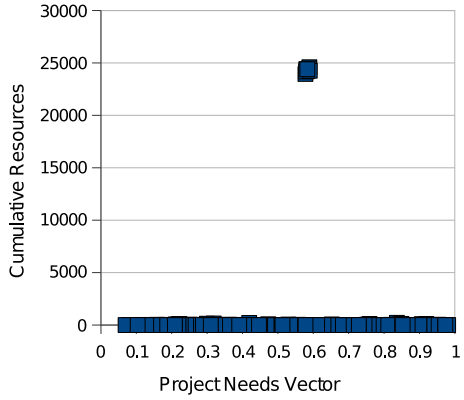
(a) Current number of consumers success metric



(b) Cumulative number of consumers success metric



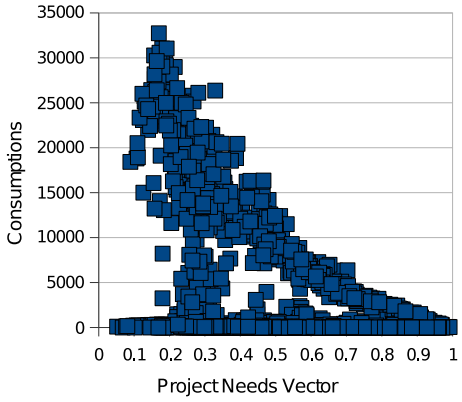
(c) Current number of producers success metric



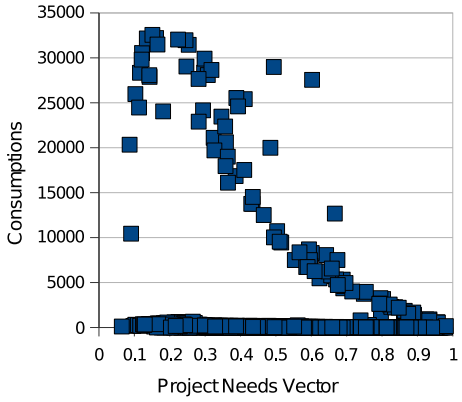
(d) Cumulative work success metric

Fig. 5.6. Project needs vectors versus cumulative resources. (a) and (b) are consumer-oriented metrics and (c) and (d) are producer-oriented metrics.

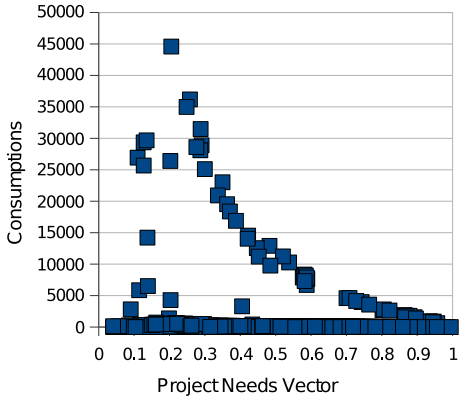
consumer-oriented projects and thus more projects with needs vectors less than 0.5 survive. Consumers then select projects for download based solely on matching interests and thus will gravitate towards consumer-oriented projects if any exist. The reason the peak consumptions are not more off center is because most agents have only a slight consumer or producer bias, as a result of how p_c and p_u are initially assigned and mapped to agent needs vectors. Thus consumer-oriented projects with very low needs vectors do not receive as many downloads as values just slightly less than 0.5 simply because there are more agents



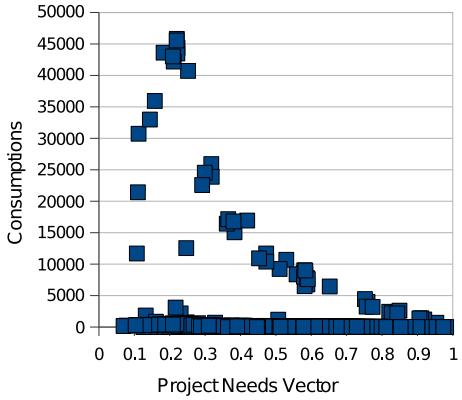
(a) Current number of consumers success metric



(b) Cumulative number of consumers success metric



(c) Current number of producers success metric



(d) Cumulative work success metric

Fig. 5.7. Project needs vectors versus cumulative consumptions. (a) and (b) are consumer-oriented metrics and (c) and (d) are producer-oriented metrics.

with mid-value needs vectors. A sample distribution of agents' needs vectors is shown in Fig. 5.8.

Fig. 5.9 shows scatterplots of project needs vectors versus success. With consumer-oriented metrics, success values, even for the best projects, are low, and the project needs vector for the most successful projects is ill-defined. The producer-oriented metrics are just the opposite. All successful projects cluster tightly around a needs vector value of 0.59, again indicating a producer bias. As mentioned earlier, this value is not more extreme

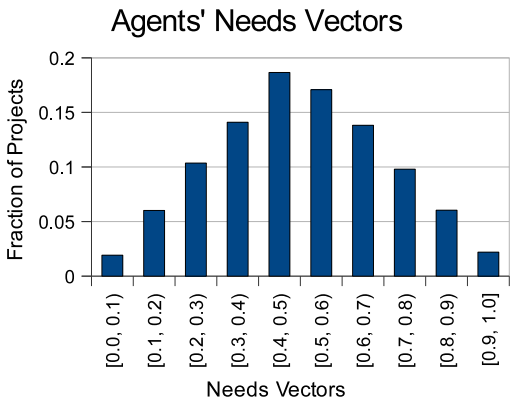
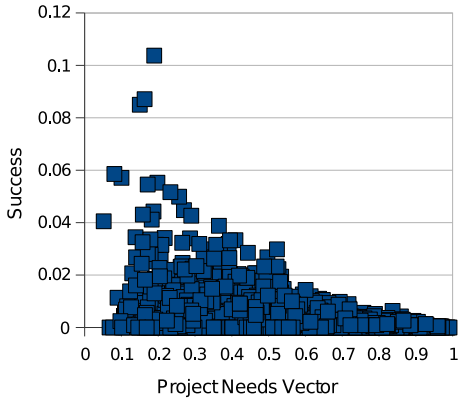


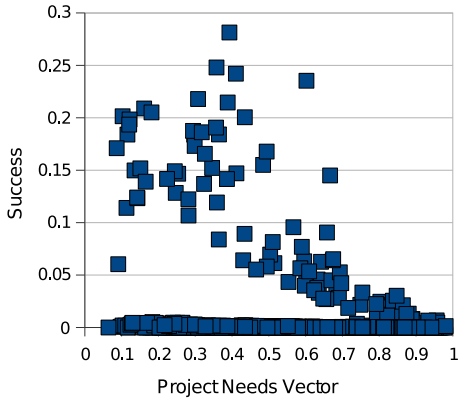
Fig. 5.8. Histogram of agents' needs vectors. Most agents have only a slight consumer (< 0.5) or producer (> 0.5) bias.

because the majority of agents have needs vectors near the center value of 0.5. The fact that all projects have extremely high success values shows the tendency of one project to dominate over all others. Essentially, it is almost impossible for a new project to become successful once there is an established successful project. This is partially due to agents always selecting the best project and is akin to the notion that popularity begets popularity. This explains why there are more non-zero values in Figs. 5.6, 5.7, and 5.9 when using consumer-oriented metrics. Consumer-oriented metrics cause a diversity in the population of surviving projects through a lack of feedback loop: producers are attracted to projects popular with consumers, while consumers are attracted to projects that match their interests. When using producer-oriented metrics, producers flock to projects that are popular with producers, which in turns makes those projects even more attractive to other producers. The positive feedback loop means only a small number of projects survive, and once a successful project is established, it becomes almost impossible for new projects to gain developers and thus increase their success.

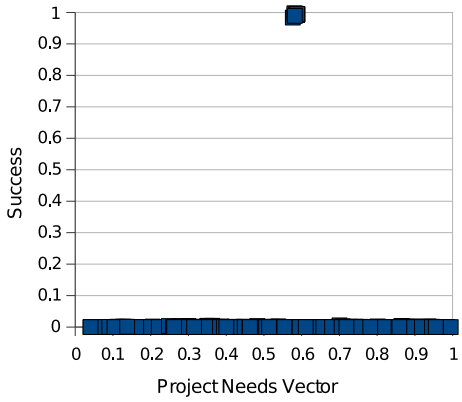
In summary, some differences are observed in surviving projects' needs vectors when using different success metrics. For example, there is more variability and a larger



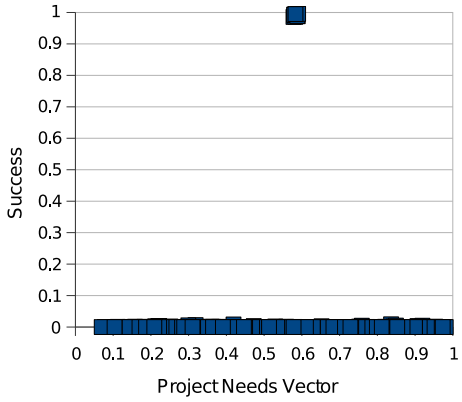
(a) Current number of consumers success metric



(b) Cumulative number of consumers success metric



(c) Current number of producers success metric



(d) Cumulative work success metric

Fig. 5.9. Project needs vectors versus success. (a) and (b) are consumer-oriented metrics and (c) and (d) are producer-oriented metrics.

number of non-zero values for cumulative resources, cumulative consumptions, and success when using consumer-oriented metrics. There is also some minor variation in peak values, although all metrics for cumulative resources and success show a producer bias while all metrics for cumulative consumptions show a consumer bias. Thus, when choosing a success metric, the exact success metric may be less important than whether it belongs to a consumer or producer category. The differences within each of these categories are minor.

5.4 CONCLUSION

Digital public goods, such as open source software, are produced by volunteers who contribute content for free. Empirical analysis shows that digital public goods experience unequal distributions of various attributes, such as the number of downloads/views of a project or the number of developers associated with a project. What causes these skewed distributions? Survey research shows that one hypothesis is that users are attracted to successful projects. However, there is no generally agreed upon definition of a successful project.

This chapter presents a simple public goods model to study the impacts of using different success metrics in the FLOSS domain. An analysis with different definitions of success shows that using different forms of measuring success has an impact on the model's output. Success may be viewed differently by consumers versus producers. Therefore the success metrics explored are categorized into consumer-oriented and producer-oriented groups. In general, differences are found between these two groups. Consumer-oriented metrics result in larger and more diverse populations of projects. Within the consumer-oriented category, there is more variation, including some cases where the *current number of consumers* metric behaves like random selection. The variability for producer-oriented metrics is much smaller. Finally, it is demonstrated that the model is producer-driven and argued that the data generated by the model when using producer-oriented success metrics has characteristics that more closely match real world data, supporting the notion that FLOSS and other digital public goods are driven by the interests of producers, not consumers.

CHAPTER 6

FLOSSSIM DESCRIPTION

Chapter 5 presented FLOSSSimple, a simple, theoretical model of the open source development process based on public goods theory. While FLOSSSimple was used to explore the effect of different success metrics and draw some general conclusions about the influence of developers and users on the FLOSS development process, the next step is to incorporate additional FLOSS processes into the model so that the model may be validated with empirical data and used for prediction purposes and scenario analysis.

This chapter presents FLOSSSim, a new model of the FLOSS development process, that borrows from existing FLOSS models and adds improvements. Section 6.1 describes the model and Section 6.2 provides details on the validation and calibration methods, as well as the setup for testing. The modeling environment is described in Section 6.3. An analysis of the model is contained in Chapter 7.

6.1 MODEL DESCRIPTION

The model universe consists of agents and FLOSS projects. Agents may choose to contribute to or not contribute to, and to consume (i.e. download) or not consume FLOSS projects.

At time zero, FLOSS projects are seeded in the model universe. These initial projects vary randomly in the amount of resources that will be required to complete them based on an exponential distribution, resulting in many small projects and few large projects. Specifically, a project's resources for completion is generated as shown in (6.1):

TABLE 6.1
Agent properties.

Property	Description	Type/Range
Consumer number	Propensity of an agent to consume (use) FLOSS.	$\mathbb{R}$ [0.0, 1.0]
Producer number	Propensity of an agent to contribute to FLOSS.	$\mathbb{R}$ [0.0, 1.0]
Needs vector	A vector representing the interests of the agent.	Each scalar in vector is $\mathbb{R}$ [0.0, 1.0]
Resources number	A value representing the amount of work an agent can put into FLOSS projects on a weekly basis. A value of 1.0 represents 40 hours.	$\mathbb{R}$ [0.0, 1.5]
Memory	A list of projects the agent knows exist.	

$$resourcesForCompletion = maxResources \cdot \mathrm{P}_{\mathrm{expPRNG}} \tag{6.1}$$

where

maxResources defines the resources required to complete the largest possible project in the model (i.e. defines the upperbound size of projects)

$\mathrm{P}_{\mathrm{expPRNG}}$ is a truncated pseudo-random number generator based on the negative exponential distribution $\lambda e^{-\lambda x}$ where $\lambda = 5$ and bounded by the range $[0.0, 1.0]$

At any time, agents may belong to zero, one, or more than one of the FLOSS projects. The simulation is run with a time step t equal to one (40 hour) work week.

Table 6.1 contains the properties of agents. Table 6.2 contains the properties of projects.

At each time step, agents choose to produce or consume based on their producer and consumer numbers, values between 0.0 and 1.0 that represent probabilities that an agent will produce or consume. Producer and consumer numbers are statically assigned when

TABLE 6.2
Project properties.

Property	Description	Type/Range
Current resources	The amount of resources or work being contributed to the project during the current time interval.	$\mathbb{R}$
Cumulative resources	The sum, over time increments, of all resources contributed to the project.	$\mathbb{R}$
Resources for completion	The total number of resources required to complete the project.	$\mathbb{R}$ [0.0, *maxResources*]
Download count	The number of times the project has been downloaded.	$\mathbb{N}_0$
Maturity	Six ordered stages a project progresses through from creation to completion.	{planning, pre-alpha, alpha, beta, production/stable, mature}
Needs vector	An evolving vector representing the interests of the developers involved in the project.	Each scalar in vector is $\mathbb{R}$ [0.0, 1.0]

agents are created and are drawn from a normal distribution, where the means and standard deviations are unknown (see Section 7.1.3.2 on page 207 for an analysis of the values used). Agents are also statically endowed with resources, representing the amount of work an agent can to contribute to projects on a weekly basis. Resources numbers are normalized to a 40 hour work week and are generated and assigned to agents based on a survey [1] that inquired how long developers spend working on FLOSS projects per week, as shown in Fig. 6.1.

All agents have memory which contains a subset of all available projects, and if producing or consuming, an agent calculates a utility score for each project in its memory.

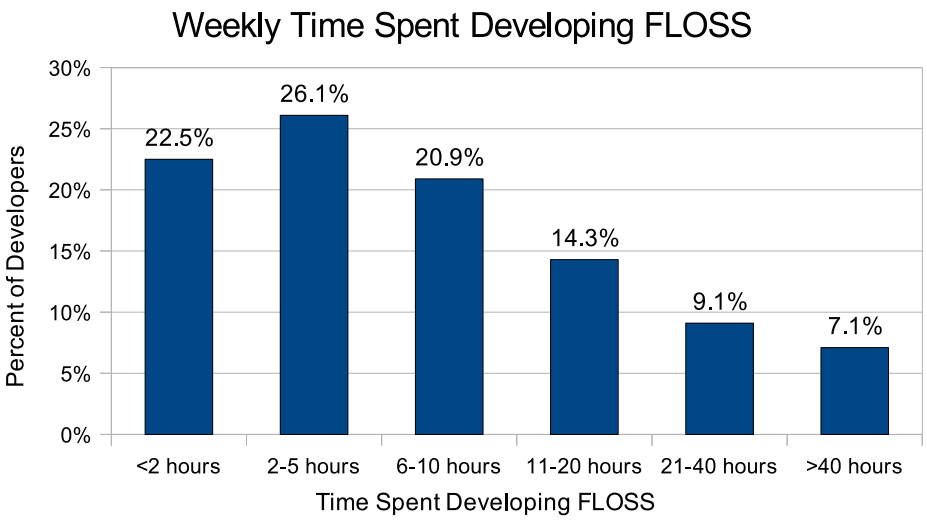


Fig. 6.1. Resources number distribution based on weekly time spent developing FLOSS. Agents' resources numbers are generated based on the distribution found in [1].

The utility function is shown in (6.2):

$$\begin{aligned} U = \; & w_1 \cdot \text{similarity}(agentNeeds, projectNeeds) \\ & + w_2 \cdot currentResources_{\text{norm}} \\ & + w_3 \cdot cumulativeResources_{\text{norm}} \\ & + w_4 \cdot downloads_{\text{norm}} \\ & + w_5 \cdot \text{f}(maturity) \end{aligned} \tag{6.2}$$

Each term in the utility function represents a factor that attracts agents to a project, where w_1 through w_5 are weights that control the importance of each factor, with $0.0 \leq w_1, w_2, w_3, w_4, w_5 \leq 1.0$ and $\sum_{i=1}^{5} w_i = 1.0$. Factors in the utility function were selected based on both FLOSS literature and a personal understanding of the FLOSS development process (see Section 3.3 for a detailed description of why these factors are considered important). Keeping it simple, a linear utility equation is used for this version of the model. The first term represents the similarity between the interests of an agent and the direction

of a project; it is calculated using cosine similarity between the agent's and project's needs vectors. The second term captures the current popularity of the project with developers and the third term the size of the project implemented so far. The fourth term captures the popularity of a project with consumers based on the cumulative number of downloads a project has received. The fifth term captures the maturity stage of the project. Values with the subscript "norm" have been normalized by dividing each project's value by the maximum value over all projects. For example, $downloads_{\text{norm}_i}$ is the *i*th project's download count divided by the maximum number of downloads that any project has received, as shown in (6.3):

$$downloads_{\text{norm}_i} = \frac{downloads_i}{\max(downloads_1, downloads_2, \ldots, downloads_{\text{numOfProjs}})} \quad (6.3)$$

where

$downloads_i$ represents the number of downloads of the *i*th project

The discreet function f maps each of the six development stages into a value between 0.0 and 1.0, corresponding to the importance of each development stage in attracting developers. The number of new developers that join a project during each stage is used as a proxy to estimate the importance of each stage in attracting new developers. Using empirical data, the number of new developers in each development stage was counted for a

subset of projects; the discovered importance of each stage is shown in (6.4).

$$\mathrm{f}(x) = \begin{cases} 0.62 & \text{if } x = \text{planning}, \\ 0.47 & \text{if } x = \text{pre-alpha}, \\ 0.27 & \text{if } x = \text{alpha}, \\ 0.23 & \text{if } x = \text{beta}, \\ 0.23 & \text{if } x = \text{production/stable}, \\ 0.0 & \text{if } x = \text{mature} \end{cases}$$
$$\text{where } x \in \{\text{planning, pre-alpha, alpha, beta, production/stable, mature}\} \tag{6.4}$$

For details on how these values were obtained, see Section 6.2.2.1.1.

Since all terms in the utility function are normalized, the utility score is always a value between 0.0 and 1.0. In addition, the square root of the utility is used instead of U when calculating the utility for projects an agent developed for or downloaded in the previous time step, representing the added utility of continuing to work on the same project due to increased familiarity with the project.

Both consumers and producers use the same utility function. This is logical, as most FLOSS developers are also users of FLOSS [13], [66]. For consumers that are not producers, arguably the terms represented in the utility function are still of interest when selecting a project. There is relatively little research published on users compared to developers of FLOSS, so it is unclear if selection criteria are different between the two groups. See Section 7.3.1 on page 230 for further exploration of consumers in the model.

It is possible that some of the terms included in the utility function are redundant or irrelevant. Part of the model calibration process is to determine which of these factors are important.

Agents use utility scores in combination with a multinominal logit equation to probabilistically select projects. The probability of selecting the *i*th project is shown in (6.5):

$$P_i = \frac{e^{\mu * U_i}}{\sum_{k=1}^{\text{numOfProjs}} e^{\mu * U_k}} \tag{6.5}$$

where

P_i is the probability of selecting the *i*th project

U_i is the utility of the *i*th project

$\{\mu \in \mathbb{R} | \mu \geq 0\}$ adjusts the level of perfect choice

The multinominal logit allows for imperfect choice, i.e., not always selecting the projects with the highest utility, and may be adjusted by changing the parameter μ. When μ is 0, all projects are chosen with equal probability regardless of each project's utility. The larger the value of μ, the greater the chance an agent will select the "best" project, that is the project with the highest utility. See Section 6.2.2.2.1 for details on how a value for μ is chosen.

Agents are limited to producing or consuming up to a maximum number of projects at each time step. When an agent is producing, the number of projects the agent is engaged in is generated from an exponential distribution with an upper cutoff, as shown in (6.6):

$$numProducing = maxNumProducing \cdot \mathrm{P}_{\text{expPRNG}} \tag{6.6}$$

where

maxNumProducing defines the upper limit for the number of projects an agent can develop for in a single time step

$P_{expPRNG}$ is a truncated pseudo-random number generator based on the negative exponential distribution $\lambda e^{-\lambda x}$ where $\lambda = 5$ and bounded by the range $[0.0, 1.0]$

By using an exponential distribution, agents will frequently be involved with a small number of projects and only occasionally develop many projects. Developer surveys have confirmed that the majority of developers work on only one or a few projects [1], [77] and the minority engage in many projects simultaneously [1], [80].

A truncated normal distribution is used when agents consume, as it is assumed that most FLOSS users use a similar number of projects and are not subject to the larger time dedication required when developers become involved in a project. Each time an agent chooses to consume, the number of projects downloaded is determined by equation (6.7):

$$numConsuming = maxNumConsuming \cdot P_{normPRNG} \tag{6.7}$$

where

maxNumConsuming defines the upper limit for the number of projects an agent can download in a single time step

$P_{normPRNG}$ is a truncated pseudo-random number generator based on a normal distribution with $\mu = 0.5$ and $\sigma = 1/6$ and bounded by the range $[0.0, 1.0]$

See Section 7.1.3.3 on page 209 for an explanation of the values used for the maximum number of projects an agent can produce or consume.

When producing, agents contribute their entire resources number to the project(s) they selected. If contributing to multiple projects during a single time step, an agent's re-

sources are distributed directly proportionally to the utility score calculated for each project. When consuming, the downloads count of each selected project is incremented.

There is no explicit formulation of communication between agents included in the model; implicitly it is assumed that agents share information about other projects and thus agents know characteristics of projects they are not currently consuming/producing. At each time step, agents update their memory. With a certain probability an agent will be informed of a project and add it to its memory, simulating discovering new projects. Likewise, with a certain probability an agent will remove a project from its memory, simulating forgetting about or losing interest in old projects. Thus, over time an agent's memory may expand and contract. See Section 6.2.2.2.4 for information on the probability of an agent updating its memory.

The model does not include an explicit switching cost in the utility function. Rather, this information is partially captured in the memory of agents, where agents are only able to contribute to the small subset of projects they are familiar with. This can be thought of as a switching cost threshold, where if an agent is not semi-familiar with a project to begin with, the cost of working on the project is considered too high. In addition, the utility of a project an agent has worked on in the previous time step is afforded a bonus by using the square root of the utility instead of the utility. This represents the added ease of staying with a familiar project over switching to another project contained in memory, for which the agent might only be semi-familiar and likely will experience startup costs when switching to.

Projects update their needs vector at each iteration using a decaying equation, as shown in (6.8).

$$pneeds_{i,t} =$$

$$\varepsilon * pneeds_{i,t-1} \tag{6.8a}$$

$$+\frac{1-\varepsilon}{resources_{i,t}} * \sum_{l=1}^{\text{numOfAgents}} (c_{l,i,t} * aneeds_l) \tag{6.8b}$$

where

$pneeds_{i,t}$ is the ith project's needs vector at time step t

$pneeds_{i,t-1}$ is the ith project's needs vector at time step $(t-1)$

$\{\varepsilon \in \mathbb{R} | 0.0 \leq \varepsilon \leq 1.0\}$

$c_{l,i,t}$ is the lth agent's contribution to the ith project at time step t

$aneeds_l$ is the lth agent's needs vector

$resources_{i,t} = \sum_{l=1}^{\text{numOfAgents}} c_{l,i,t}$

The ith project's vector at time t, $pneeds_{i,t}$, is partially based on the project's needs vector at time $t-1$ (6.8a) and partially on the needs vectors of the agents currently contributing to the project (6.8b). The rate of decay is controlled by ε. An agent's influence on the project's needs vector is directly proportional to the amount of work the agent is contributing to the project with respect to other agents working on the same project at time t. This represents the direction of a project being influenced by the developers working on it, with core developers having a larger influence than peripheral developers in steering the project. The use of a decaying equation allows projects to change their direction based on the agents contributing to them while at the same time maintaining inertia, based on work already completed

on the project. The speed at which a project is able to change is adjusted by changing ε. See Section 6.2.2.2.5 for details on how the value of ε is set.

Projects are assigned to maturity stages based on the percent of the project that is complete. Setting the thresholds for the development stages is based on empirical data; how the boundaries were determined is described in Section 6.2.2.1.2. Projects are assigned to development stages based on (6.9).

$$\mathrm{d}(x) = \begin{cases} \text{planning} & \text{if } 0.0 \leq x < 0.11, \\ \text{pre-alpha} & \text{if } 0.11 \leq x < 0.25, \\ \text{alpha} & \text{if } 0.25 \leq x < 0.53, \\ \text{beta} & \text{if } 0.53 \leq x < 0.78, \\ \text{production/stable} & \text{if } 0.78 \leq x < 0.96, \\ \text{mature} & \text{if } 0.96 \leq x \leq 1.0 \end{cases}$$

where x is the fraction of resources completed on a project, i.e., $\frac{cumulativeResources}{resourcesForCompletion}$

(6.9)

For example, a project in the model will be considered in the alpha stage if between 25% and 53% of the work has been completed.

When created, projects' cumulative resources may be initialized to non-zero values, representing projects that have accumulated work before being released as FLOSS. These values are assigned based on empirical data, further outlined in Section 6.2.2.1.4.

New projects are created and added to the model at each time step, representing the new projects that are constantly being added to the open source community. The rate of

creation of projects is based on empirical data from SourceForge and scaled to match the size of the simulation. See Section 6.2.2.1.3 for further details on this data.

A project is considered complete when its cumulative resources is equal to its resources for completion. Completed projects are not removed from the model. While they are not eligible for further contributions by developing agents, these projects may still be downloaded by users.

Finally, projects are assumed to be completely independent of one another; there are no explicitly modeled links between projects.

6.2 Model Evaluation

To aid with evaluating the model, a framework called pattern-oriented modeling is employed during both the calibration and validation stages. Pattern-oriented modeling attempts to address concerns with complexity and uncertainty in bottom-up modeling [190], [191], [192], [193], [194], such as agent-based modeling. The idea is to identify patterns, such as emergent properties, that capture information about the internal structure and processes of the system being modeled [194]. Models that are able to match important patterns are also more likely to be structurally realistic. The pattern-oriented modeling framework recommends matching multiple patterns from different hierarchical levels and scales, especially when modeling complex systems where matching a single pattern may be insufficient to demonstrate that the model is structurally realistic [194]. The ability of a model to match multiple patterns simultaneously increases confidence that the model is structurally realistic [193]. Structurally realistic models may not only reproduce the real world behavior of a system but also their innerworkings perform in a manner that is similar to that which occurs in the real system [195], thus lending credibility to the model. Indeed, models that

are structurally realistic have been able to make accurate secondary predictions for which they were not originally designed [194].

In general there is a relationship between a model's payoff and a model's complexity. Oversimplified models may not be able to duplicate the characteristics of the real system because they lack essential mechanisms found in the domain being modeled [194]. On the other hand, overly complex models may be difficult to understand and gain knowledge from, may be sensitive to parameter changes, etc. [194]. Thus, a balance must be sought between these two extremes. Pattern-oriented modeling strives to find an ideal level of complexity for a model by fostering the elimination of components that do not significantly contribute to matching important patterns while retaining components that are necessary to reproduce important phenomena [194]. In other words, pattern-oriented modeling attempts to reduce the complexity of a model by eliminating extraneous components until all that remains is what is necessary to reproduce the key characteristics of the system.

Finally, pattern-oriented modeling assists with the model calibration process by addressing parameter uncertainty. First, by encouraging models to be structurally realistic, the resulting models will likely be less sensitive to parameter uncertainty [196]. Secondly, pattern-oriented modeling helps with the process of estimating unknown or uncertain values by finding combinations of parameters that match multiple patterns simultaneously. In instances where the parameter state space is large and only one pattern is being matched, there may be a large number of parameter combinations that work equally well in reproducing the pattern. Matching multiple patterns simultaneously, especially when the patterns are selected from different parts of the system and are not directly related, is likely to be non-trivial; evidence has shown that matching multiple patterns can greatly reduce the number of parameter combinations that are considered acceptable (e.g., see [197]). If there are no

parameter combinations that allow a model to reasonably match the chosen patterns, it may indicate that the model is not structurally realistic or that key mechanisms or processes from the system are not included in the model [197]. This in turn reinforces the importance of creating structurally realistic models when applying pattern-oriented modeling [193]. Because all patterns occur simultaneously in the real system, it is more likely that the model will be able to match all the patterns if the model's organization and processes are similar to that of the actual system being modeled.

6.2.1 Validation Method

In accordance with the recommendations of pattern-oriented modeling, three patterns, in the form of emergent properties, are chosen to validate FLOSSSim. If the model is able to reproduce all three patterns simultaneously, confidence in the validity of the model is increased. As recommended by the pattern-oriented modeling framework, each of the three emergent properties is from a different area of the system being modeled, thus providing a diverse set of patterns that is expected to be nontrivial to match and therefore useful for calibration and validation purposes.

The following three emergent properties were chosen to validate the model:

Distribution of projects in development stages: In December 2004, [117] crawled SourceForge and counted the number of projects in the planning, pre-alpha, alpha, beta, production/stable, and mature development stages. Using data from FLOSSmole [156], the distribution of projects in development stages was independently found to be very similar in June 2009, as shown in Fig. 6.2. The 2004 data from [117] is used to calibrate the model. A test of the model's ability to predict the 2009 data when calibrated with the 2004 data is performed in Section 7.1.2.1 on page 195.

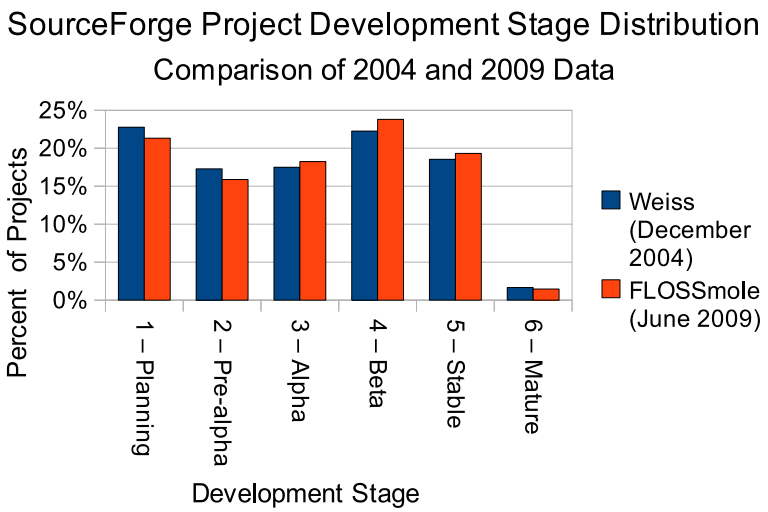


Fig. 6.2. The percentage of SourceForge projects in six development stages. Data from a crawl performed by [117] in December 2004 is compared to data mined from FLOSSmole's June 2009 crawl. Although separated by 4.5 years, the percentage of projects in each stage is very similar, indicating this distribution has remained relatively constant over time.

Number of developers per FLOSS project: This is a highly skewed distribution, potentially making it difficult and therefore useful to match for validation purposes, with most projects having a small number of developers and only a few projects having a large number of developers. Data is available from [117]'s December 2004 crawl of SourceForge. Using data from FLOSSmole [156], the distribution of developers per project was independently found to be very similar in June 2009, as shown in Fig. 6.3, with the main difference being a larger number and more extreme outliers (not shown in Fig. 6.3) in the 2009 data[1]. The 2004 data from [117] is used to cal-

[1]In the June 2009 FLOSSmole data, 19 projects had over 100 developers involved, with the record set by tikiwiki (http://sourceforge.net/projects/tikiwiki/) which had an astounding 428 developers registered with the project. However, fewer than 0.1% of the projects had more than 41 developers, adding confidence that these projects are outliers. FLOSSSim is concerned with modeling the normal cases rather than the exceptions.

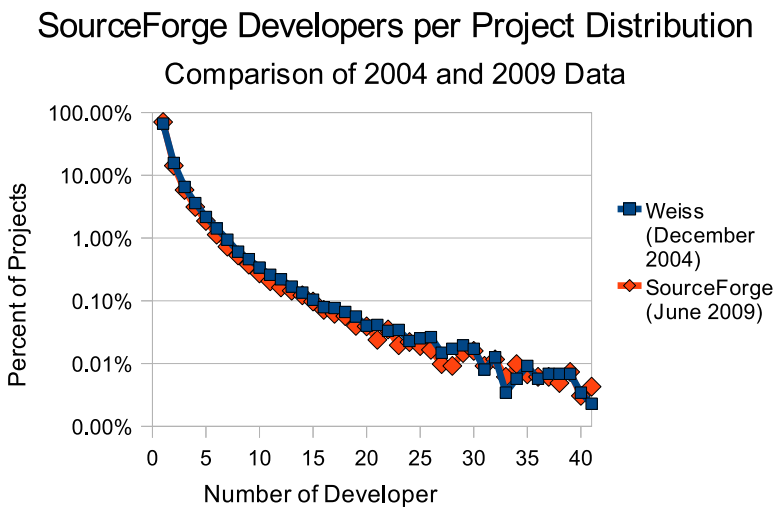


Fig. 6.3. The percentage of SourceForge projects with N developers. Data from a crawl performed by [117] in December 2004 is compared to data mined from FLOSSmole's [156] June 2009 crawl. Not shown are the fewer than 0.01% of projects in the 2009 data that have more than 41 developers. Although separated by 4.5 years, the percentage of projects with N developers is very similar.

ibrate the model. The model's ability to predict the 2009 data when calibrated with the 2004 data is explored in Section 7.1.2.1 on page 195.

Number of FLOSS projects per developer: The bulk of developers work on one or only a few FLOSS projects while a few developers are involved in many projects. Data is used from a survey [1] conducted from February through April 2002 of 2784 developers. Although this data is relatively old, it comes from one of the largest, well-conducted surveys, involving many projects and developers; no equivalent surveys have been conducted since.

By creating a model that mimics a number of key patterns based on empirical data, confidence is derived about the model.

6.2.2 Calibration Method

The model includes a number of parameters that must be assigned values. A subset of these can be set to likely values based on statistics gathered from surveys or mined from FLOSS repository databases. Parameters that are difficult or impossible to measure must instead be estimated. As previously mentioned in Section 6.2, applying pattern-oriented modeling can assist in estimating values by finding combinations of parameters that are able to simultaneously match multiple patterns. The parameter values that perform well may then be used as estimates for the real, but unmeasurable, values.

Section 6.2.2.1 discusses values that are set based on statistics mined from data repositories. Section 6.2.2.2 outlines parameter values that are set based on literature, logic, and personal experience while developing the model. Finally, Section 6.2.2.3 explains how genetic algorithms are used to explore the state space, in conjunction with pattern-oriented modeling, to efficiently find values for which there is no practical method to estimate. These evolved parameters offer insight into the FLOSS development process and are further discussed in Section 7.1.3 on page 200.

6.2.2.1 Mined Values

Using the data sources outlined in Chapter 4, data mining was performed to extract information necessary to configure the model. The following sections describe the mining process used to obtain certain values used in the model.

6.2.2.1.1 Maturity Stage Importance: The model depends on knowing the importance of each of the six maturity stages in regards to attracting developers to projects. Are developers drawn to projects in the early stages of development, such as planning and pre-alpha, when main design decisions are being made and core code still

needs to be developed, or do developers prefer later stages, when the software has already proven itself as stable? [45] argues that developers are attracted to early development stages due to the longevity of the code, and thus long-term programmer recognition, that becomes the cornerstone of future development. On the other hand, developers may not want to waste their time on a project that, several months after contributing to it, becomes inactive without ever creating usable software. In this case, developers may be interested in later stages, such as beta and production/stable, at which point the software has already demonstrated some functionality and thus contributions at this point may be seen as unlikely to be wasted, since the project has already proven its ability to survive.

Counting the number of developers involved in a project at each maturity stage initially appears to be an acceptable proxy for measuring the importance of maturity stages. More desirable development stages will have more developers involved. Unfortunately, there is an inherent flaw in this metric. Sites like SourceForge require users to create accounts, after which developers can use their accounts to join projects. The problem with blindly counting the number of developers registered with a project for each stage is that this count will include both active and inactive developers. That is, it is relatively common for a developer to join a project but then, either immediately or sometime in the future, provide no contributions. Inherently, there is a bias towards joining a project – one must join a project to, for example, commit code – but there is no reason to remove one's self from a project when no longer interested in contributing. [48] includes several examples where counting the number of developers causes a project to appear active with a steady development team when, in fact, the project was abandoned long ago. For the work performed in [48], the metric is adapted to only count active developers by checking SCM logs to verify a developer has made a contribution within a certain timeframe. Since the focus of

this research is understanding what maturity stages attract developers to a project, the metric is further modified to count only new developers, where new developers are those that have never worked on the project before. If a developer leaves for a period of time and then later returns to the same project, that developer will only be counted the first time he/she joins and not when he/she returns. This metric then helps gauge at what development stage projects become desireable enough to join.

Obtaining the number of new developers per maturity stage involved combining data from multiple sources. Since the goal was to determine the relative importance of all six development stages, it was necessary to select projects that had progressed through as many stages as possible. Starting with the projects in the FLOSSmole database [156], after removing projects that were listed in multiple stages at the same time (considered dirty data since projects should only have a single development status at any given time), there were zero projects that had progressed through all six stages and only two that had gone through five stages. This made it necessary to lower the threshold to projects that had progressed through four or more stages in order to obtain a sufficient number of projects to study, resulting in 76 projects. The dates each of these projects changed maturity stages were extracted from the FLOSSmole database, but the FLOSSmole database did not include enough information to determine when new developers joined a project. While the FLOSSMetrics data included the SCM data necessary to count the number of new developers, none of the 76 FLOSSmole projects were included in the FLOSSMetrics data set. Therefore, CVS, SVN, and git were used to manually download SCM logs for these projects and then CVSAnalY2 [165], a tool used by FLOSSMetrics, was used to convert the SCM logs into relational database tables. The resulting databases could then be queried to determine

the number of new developers that joined a project bounded by the dates, extracted from the FLOSSmole database, that indicated when a project was in each development stage.

For each project, the new developer counts were normalized by finding the stage with the maximum number of new developers and dividing all stages by this number, as shown in (6.10).

$$nd_{\text{norm}_{i,j}} = \frac{nd_{i,j}}{\max(nd_{i,\text{planning}}, nd_{i,\text{pre-alpha}}, \ldots, nd_{i,\text{mature}})} \tag{6.10}$$

where

$nd_{i,j}$ is the number of new developers for the ith project in the jth development stage

$\{i \in \mathbb{N}_1 | 1 \leq i \leq \text{number of projects}\}$

$j \in \{\text{planning, pre-alpha, alpha, beta, production/stable, mature}\}$

For each development stage, the normalized values were averaged across the 76 projects, resulting in a normalized importance value for each of the development stages. Projects that had never been in a particular development stage were omitted when calculating the average for that stage only. These values were used to define the discrete function f as shown in (6.4).

By definition, all the developers are considered new when a project is first released as open source, resulting in a high importance value for the planning stage. The data show that in general, fewer and fewer new developers join as a project matures. Unfortunately, in the data set there were only four projects that progressed to the mature stage, and none of these projects gained new developers once in this stage. As a result of the small number of mature projects, the zero weight reported for this stage might not be accurate.

6.2.2.1.2 Maturity Stage Thresholds: In the model, there must be a mechanism to determine the maturity stage of a project. In the real world, this is a human-assigned value that is updated by a project administrator to convey information about how mature or complete the software is. The value is ideally chosen by someone who is entrenched in the project, most likely a core developer, and in theory reflects the level of functionality, stability, etc. of the project. Inherently, there appears to be a connection between development stage and the amount of work completed on a project. A project listed in the alpha stage, for example, is generally understood to have limited functionality, whereas software in the beta stage is expected to have the majority of the functionality implemented, even though it may still include many bugs. There will of course be some variability from project-to-project as to exactly what each development stage represents, but the general idea is that the higher the development stage, the more work that will have been completed on a project. Because of this, FLOSSSim approximates the human-assigned maturity values by calculating the percentage of a project that is complete and assigning it a maturity stage based on this percentage.

To map development stages into percent complete thresholds, the amount of work that was complete when real world projects moved maturity stages is examined. The results should be averaged across many projects to provide an estimate on the amount of work typically completed in each stage, from which point thresholds in the model can be set. FLOSSSim tracks the amount of work performed on a project based on man hours contributed to a project (see "resources number" in Table 6.1 and "current resources," "cumulative resources," and "resources for completion" in Table 6.2). Unfortunately, empirical data tracking the number of hours contributed to projects is not available. Instead, code commits, which are tracked by SCM software and therefore available for many projects, are

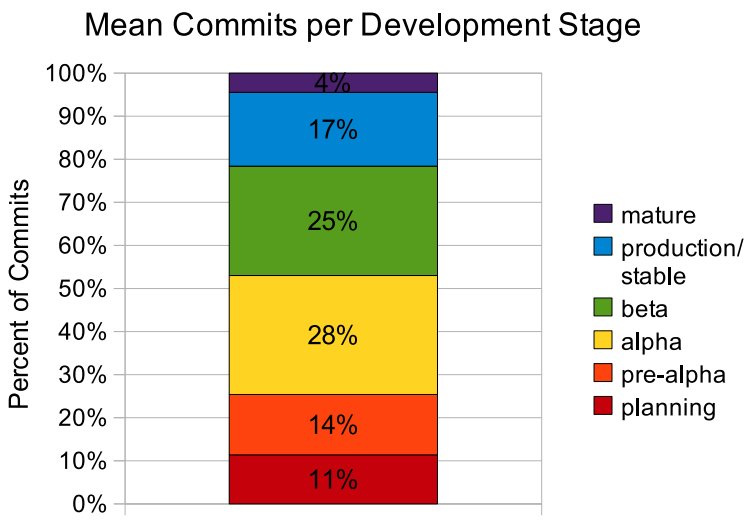


Fig. 6.4. Mean percentage of code commits that occur in each development stage.

used as a proxy for work completed. Essentially both man hours and code commits measure work completed on a project, but the former represents a fine-grained measurement while the latter represents a more coarse measurement. That is man hours represent the raw time spent working on a project whereas commits represent meaningful increments of work.

To calculate the maturity stage thresholds based on the percent complete of a project, the percentage of code commits that occurred in each development stage for projects on SourceForge that had progressed through four or more development stages was determined. The dates projects changed stages was ascertained using the FLOSSmole database [156]; CVSAnalY2 [165] was then used to build database tables from each project's SCM logs, which were subsequently queried to determine the number of commits in each stage. The mean percentage of commits that occur in each stage is shown in Fig. 6.4. This information is used to define (6.9).

Not surprisingly, the data show that projects upgrade rapidly to the alpha stage once some minimal functionality has been obtained through a small number of commits. The

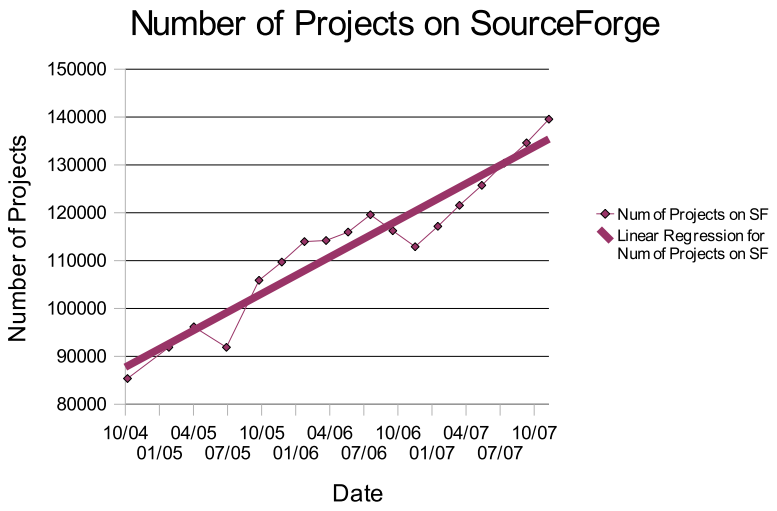


Fig. 6.5. The number of projects on SourceForge with respect to time.

majority of work is performed while projects are in the alpha and beta stages. Finally, as projects mature the number of commits is reduced as projects shift from a development phase, where adding new features and functionality is the predominant activity, to a maintenance phase, where the focus is mostly on bug fixes, support, and upkeep of the code.

6.2.2.1.3 New Project Creation Rate: To determine the number of projects to add to the model at each time step, the growth rate of projects on SourceForge is analyzed. Using the FLOSSmole database [156], the date of each crawl along with the number of projects included in each crawl is calculated. Clean data was available from October 2004 through December 2007. Although FLOSSmole attempts to crawl SourceForge every 60 days, there are periods when more than 60 days elapsed without a crawl, including from October 2004 through October 2005, when crawls occurred quarterly. A plot of the date versus number of projects on SourceForge is contained in Fig. 6.5. The equation for

the best-fit line is shown in (6.11):

$$y = 1242x + 88098 \tag{6.11}$$

where

x is the number of months offset from October 2004 (i.e., 10/04 is 0, 11/04 is 1)

For the regression line, $R^2 = 0.92$, indicating that the line is a good fit and supporting that projects are added to SourceForge at a roughly linear rate, namely around 1242 projects per month. The occasional decrease in the number of projects appears to occur when SourceForge performs some level of housekeeping and removes dead projects, although SourceForge's policy about the removal of projects does not appear to be publicly documented (i.e. frequency, criteria for removing a project, etc.).

FLOSSSim is normally run with a reduced set of projects, in which case the rate at which projects are added to the simulation is appropriately scaled.

6.2.2.1.4 New Project Cumulative Resources and Maturity: Not all FLOSS projects start from scratch as open source. Many projects are first partially developed and then released under an open source license. The development stage of projects when first added to SourceForge, as mined from the FLOSSmole database, is shown in Table 6.3. When creating projects, FLOSSSim initializes the cumulative resources and maturity according to this distribution. The cumulative resources is set to the bottom threshold of each development stage, i.e., if a project is created in the alpha stage, then the cumulative resources is set to 25% of the resources for completion because 11% and 14% of the resources are contributed in the planning and pre-alpha stages respectively (see Fig. 6.4).

TABLE 6.3
Development stage of projects when first added to SourceForge, based on data mined from the FLOSSmole database (see [156]).

Initial Development Stage on SourceForge	Percent of Projects
planning	26.1%
pre-alpha	16.6%
alpha	18.1%
beta	22.4%
production/stable	15.6%
mature	1.2%

6.2.2.2 Defined Parameter Values

A subset of model parameters cannot be directly measured and instead must be assigned values based on literature, logic, and experience. A summary of these model parameters and their assigned values is contained in Table 6.4. Details describing how individual parameters are assigned values are contained in the following subsections.

6.2.2.2.1 μ: μ controls the level of perfect choice in the model when agents are selecting projects (see (6.5)). When $\mu = 0$, all projects are selected with an equal probability. As μ approaches ∞, the probability of selecting the project with the highest utility increases. Unfortunately, it is impossible to directly measure μ and therefore this value must be estimated. As in the real world, agents in the model are only aware of a small subset of all existing projects. Since this subset is relatively small (although the size varies by individual), it is expected that individuals will be able to collect enough information to make near-perfect decisions. Therefore, μ is assigned the relatively high value of 36, a value that causes agents to make well-informed decisions. See Section 7.2.1 on page 222 for details on how this value was selected.

TABLE 6.4
Defined FLOSSSim model parameters.

Parameter	Defined Value	Description
μ	36	Controls agents' level of perfect choice. See (6.5).
Needs vector dimension	3	The dimension of projects' and agents' needs vectors.
Starting memory size	5	The initial number of projects an agent is aware of when the agent is created.
Memory change probability	0.065	The probability of an agent, at each time step, adding and/or removing a project from its memory.
ε	0.5	Decaying constant that controls how quickly a project changes its needs vector to accommodate agents working on the project. See (6.8).
Number of agents	389	The number of agents at the start of a model run.
Number of projects	1024	The number of projects at the start of a model run.
Maximum resources	10,000	The amount of work required to complete the largest possible project in the model. Controls the maximum size of projects.

6.2.2.2.2 Needs Vector Dimension: There is no way to directly derive from empirical data the correct dimensionality for the needs vector, since the needs vector is already an abstraction of the real world used to represent the interests and directions of agents and projects respectively. Therefore, this value must be estimated. When setting the dimensionality, there is a tradeoff. If the value is set too low, this will result in insufficient variation in projects and agents, making the similarity very close between most projects and agents. Setting the dimensionality too high is not expected to increase the variation in a manner that causes the model to perform more realistically, but it will increase the computation time for calculating the similarity. Since similarity is calculated frequently by

the model, even a small increase in time to compute the similarity can have a large effect on overall execution time. Therefore, the goal is to use the minimum value that provides for sufficient variation in agents' and projects' needs.

One and two dimensional vectors are expected to provide insufficient variation. Experience with the model shows that values greater than two are able to reproduce the validation patterns. A sensitivity analysis, described in Section 7.2.2 on page 225, demonstrates that values greater than three do not result in better fitness. Therefore, the needs vector dimension is set to three.

6.2.2.2.3 Starting Memory Size: Developers and users can only participate in projects that they are aware of. However, it is unknown how many FLOSS projects the average developer or consumer knows about. For the purpose of FLOSSSim, when agents are created they are made aware of a subset of projects; over time, an agent may learn about new projects, adding them to its memory, and forget and/or reject other projects, removing them from its memory. The number of projects an agent's memory is seeded with when the agent is created must be estimated. For FLOSSSim, this value is set to five, as determined through a sensitivity analysis outlined in Section 7.2.3 on page 225.

6.2.2.2.4 Memory Change Probability: At each time step agents may learn about new projects, adding them to the subset of projects they can select from, or forget about or reject projects that are already in their memory. The probability of individuals discovering or forgetting projects cannot be determined using available empirical data. A sensitivity analysis, contained in Section 7.2.4 on page 227, shows the model performs well when the change probability is 0.065.

6.2.2.2.5 ε: ε controls the amount a project's needs vector at time t is influenced by the project's needs vector at time $t-1$ versus the needs vector of the devel-

opers working on the project at time t (see (6.8)). There is no method to measure ε from empirical data and therefore this value must be estimated. A value of 0.5 is chosen, which should allow projects to be agile and rapidly adapt as the developer population changes while still providing a level of inertia to the project that comes from the work performed by earlier developers.

6.2.2.2.6 Number of Agents and Projects: There are two goals to consider when setting the number of agents and projects in the model runs. First is the ratio of agents to projects. If this ratio is too high, too much work will be accomplished and projects will progress too rapidly. If the ratio is too low, not enough work will be completed and all projects will be stagnant. The correct ratio is also affected by the producer and consumer number means and standard deviations. For example, a higher producer number mean causes agents to more frequently develop, meaning the same number of agents will be able to complete the same amount of work in less time. For an analysis of the evolved producer and consumer number means and standard deviations, see Section 7.1.3.2 on page 207.

A second goal is to minimize execution time. It is desirable to scale down the absolute number of agents and projects to values that still allow the model to match the real world phenomena but that do not consume excessive computing resources with no value added. Although execution time is secondary to the model's primary purposes, the use of evolutionary computation in the development of FLOSSSim results in the model being run millions of times, making execution time important nonetheless.

A sensitivity analysis, contained in Section 7.2.5 on page 227, is used to determine both the optimum agent-to-project ratio and the absolute values that are necessary in order for the model to match empirical data. This results in model runs starting with 389 agents and 1024 projects, with more projects being created during the execution of the model.

6.2.2.2.7 Maximum Resources: The value of the maximum resources sets the size of the largest possible project in the model. A sensitivity analysis shows that the model performs well over a wide range of values and therefore 10,000 is chosen as a reasonable value for the model. This means that the largest possible project that could be created in a simulation run would take 10,000 40 hour work weeks to complete, the equivalent of 48 people working full time on a project for 4 years.

6.2.2.3 Genetically Evolved Values

Unfortunately, not all parameters in the model can be directly measured or intelligently assigned values; values for these parameters must instead be sampled from bounded ranges, using pattern-oriented modeling to guide the search for well-performing values. The evolved values may then be examined to gain a better understanding of the FLOSS development process. Descriptions and bounded ranges of the model parameters that are evolved are contained in Table 6.5.

For the parameters that must be estimated, a search of the parameter space must be performed to find the combination(s) that allow the model to most closely match the empirical data. Due to the large state space, an exhaustive search is not possible. Since genetic algorithms are known to perform well in high dimension, stochastic, non-linear spaces [198], genetic algorithms are employed to find near-optimal parameter sets that result in the model's output closely matching the empirical data. This is done as follows: an initial population of 30 model parameter sets is created, with each individual parameter assigned a random value drawn from a uniform distribution that is bounded by estimated upper and lower limits for the given parameter (see the range column in Table 6.5). The model is run with each of the parameter sets and a fitness score is calculated based on the similarity of the generated versus empirical data. The parameter values from these sets are then mutated or

TABLE 6.5
Descriptions and value ranges for model parameters that are evolved. Values for these parameters cannot be easily measured or estimated. Pattern-oriented modeling, in conjunction with genetic algorithms, is used to estimate these parameter values by finding combinations of parameters that result in the model producing data that closely matches empirical data.

Parameter	Description	Range
w_1	Similarity weight; the importance of similarity between an agent and a project when selecting a project.	[0.0, 1.0]
w_2	Current resources weight; the importance of the amount of work currently being done when selecting a project.	[0.0, 1.0]
w_3	Cumulative resources weight; the importance of the amount of work already completed when selecting a project.	[0.0, 1.0]
w_4	Downloads weight; the importance of the popularity of a project with consumers when selecting a project.	[0.0, 1.0]
w_5	Maturity weight; the importance of a project's development stage when selecting a project.	[0.0, 1.0]
maximum number consuming	The maximum number of projects an agent can use during a single time step.	[3, 12]
maximum number producing	The maximum number of projects an agent can develop for during a single time step.	[3, 15]
producer number mean	The mean value of the normal distribution used for generating agents' probability of producing.	[0.0, 1.0]
producer number stdev	The standard deviation of the normal distribution used for generating agents' producer numbers.	[0.0, 1.0]
consumer number mean	The mean value of the normal distribution used for generating agents' probability of consuming.	[0.0, 1.0]
consumer number stdev	The standard deviation of the normal distribution used for generating agents' consumer numbers.	[0.0, 1.0]

crossed-over with other parameter sets to create a new generation of model parameter sets, with a bias for selecting parameters sets that resulted in a high fitness; then the new generation of parameter sets are evaluated and the process is repeated. This repetition continues

until individuals meet a fitness threshold or a maximum number of generations is evolved[2]. In this case, a genetic algorithm is being used for a stochastic optimization problem for which it is not known when a global optimum is found. Genetic algorithms are appropriate for finding well-performing solutions in a reasonably brief amount of time and frequently outperform other optimization techniques, such as random search and hill climbing, when applied to non-linear, chaotic, or stochastic landscapes [198]. Reviewing the values of the best performing parameters will help identify which factors are important/influential in the open source software development process.

The fitness function chosen for the genetic algorithm is based on the sum of the square of errors between the simulated and empirical data, as shown in (6.12):

$$fitness = 1 - \frac{\text{sum of square of errors}}{\text{maximum possible sum of square of errors}} \tag{6.12}$$

Since there are three fitness values calculated, one per empirical data set, the three fitness values are averaged to provide a single value for comparison purposes.

6.2.3 Setup for Testing

Since the model includes stochastic components, multiple runs with a given parameter set are performed and the results averaged. To determine the number of runs necessary, 256 parameter sets were chosen randomly and each of these sets was evaluated 32 times. Using normal probability plots, it was determined that fitness values for each parameter set were normally distributed. From these runs the standard error of the mean for the model fitness was calculated, which provides information on the probability of finding the real mean fitness of the population based on the sample size. The probability of obtaining the

[2]For the results presented, up to 4096 generations were evaluated.

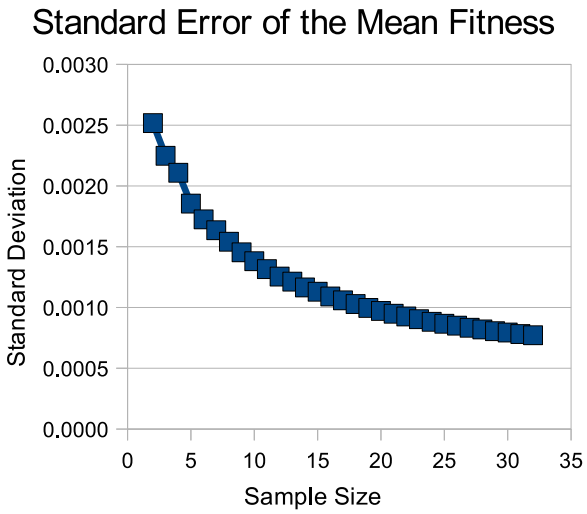


Fig. 6.6. Standard error of the mean fitness based on 256 random parameter sets.

actual mean of the population increases at a diminishing rate with increases in sample size. The standard error of the mean fitness is shown in Fig. 6.6. While an increase in sample size results in a better estimate of the population's true mean, it also increases the computation time necessary to evaluate each parameter set. Therefore, a balance must be found between these two conflicting issues. Based on Fig. 6.6, eight runs are performed because 1) increases in sample size beyond eight result in only a minor decrease in standard deviation and 2) via the genetic algorithm, important parameter sets (i.e. those with high fitness scores) are likely to be copied to a subsequent generation and therefore reevaluated multiple times across generations, decreasing the chances of a poor parameter set with a few lucky runs being incorrectly categorized as a good fit. As a result of these findings, each parameter set is evaluated eight times when evaluating the model and the averaged results over eight runs are then compared to the empirical data.

As empirical investigations of FLOSS evolution note, it takes approximately four years for a project of medium size to reach a mature stage [80]. To accommodate large

projects and allow the simulation sufficient time to stabilize from startup conditions, the model's performance is evaluated after 250 time steps, with a time step t equal to one week, resulting in a simulated period of just under five years. All metrics are gathered immediately following the 250th time step.

The results of running the model as outlined here are presented in Section 7.1 on page 190.

6.3 Modeling Environment

The following sections provide details of the modeling environment. The criteria for selecting the modeling platform are contained in Section 6.3.1, the hardware used for the simulation runs is described in Section 6.3.2, and the verification techniques employed when developing the code are disclosed in Section 6.3.3.

6.3.1 Modeling Platform

After performing a cursory review of potential modeling environments supporting agent-based modeling, three toolkits were selected as having the most promise:

NetLogo: (http://ccl.northwestern.edu/netlogo/) This is the modeling environment FLOSSSimple was implemented in. In the spirit of logo programming languages, it is high-level, but proprietary, and aims to be simple to use yet allow for rapid development. It includes support for visualization and is itself written in JAVA, providing for platform independence.

Repast: (http://repast.sourceforge.net) The Recursive Porous Agent Simulation Toolkit is an open source suite of tools that support modeling and simulating. Repast has been

reimplemented in multiple languages, meaning a version is available for most platforms. Programming in Repast may be done in logo-like languages or using traditional programming languages (e.g., JAVA, C++) using Repast as a software library.

MASON: (http://cs.gmu.edu/~eclab/projects/mason/) The Multi-Agent Simulator Of Neighborhoods (or Networks... or something...) is an open source JAVA library. As such, it is platform independent, and models using MASON are implemented in JAVA.

All three toolkits were originally developed in academic environments: NetLogo was developed at Northwestern University, Repast at the University of Chicago, and MASON at George Mason University.

When selecting from these options of modeling platforms, the following important aspects were taken into consideration:

Execution speed: Because the model will be executed literally tens of millions of times during development, execution speed is a very important factor. NetLogo is inherently slow, as the logo language is interpreted and the interpreter itself runs in JAVA, another slow language because it is also interpreted. Both Repast and MASON aim to be fast, even though both are also written in JAVA. Although JAVA is at an inherent disadvantage compared to natively compiled languages, the MASON team points out that well-written JAVA can execute astoundingly fast [199] and a design point of the MASON library is to efficiently support large simulations [199]. Indeed, some comparisons have shown that while Repast may be a more complete modeling library, models implemented in MASON execute faster [200].

Parallel execution support: Again, because of speed concerns, a simulation environment that can take advantage of multiple processors is preferred. NetLogo does not support multi-threaded programming [201]. Both Repast and MASON, because they are JAVA programming libraries, support threads through JAVA's native support for threads.

FLOSS license: A modeling platform that is open source itself provides all the benefits of FLOSS, including the ability to fix bugs oneself rather than waiting for official releases[3]. NetLogo is closed source, but both Repast and MASON use open source licenses.

In addition, MASON offered several other major advantages over the competition. Namely, MASON natively supports checkpointing so that simulations may be stopped and restarted. This is invaluable when performing long runs that may last weeks, as it allows restarting from the last checkpoint should a failure occur (e.g., hardware failure, power outage). This also allows jobs to be migrated from one computer to another as resources become available. MASON also guarantees identical results regardless of the execution platform, meaning that migrating a job mid-execution has no effect on the results. In addition, MASON is designed to allow the attachment and detachment of modules to running models; this allows, for example, a model to be run quickly without visualization, checkpointed, and then restarted from the checkpoint with a visualization module attached, or vice versa. In addition, the the Evolutionary Computation Lab at George Mason University

[3]In the case of MASON, I not only discovered a bug and managed to track it back to the MASON code, but I was also able to fix the error, document it, and submit a proposed patch to the developer mailing list. The fix has been incorporated into the project.

that produced MASON has also written an evolutionary computation toolkit called ECJ[4]. ECJ can be used with MASON to provide the genetic algorithms used in this research. Like MASON, ECJ is optimized for speed and is also under an open source license[5]. ECJ also provides built-in thread support for evaluating and breeding multiple individuals in a generation concurrently. Additional checkpointing abilities are also built into ECJ.

Based on the advantages MASON had over the other platforms, MASON was selected for implementing FLOSSSim.

6.3.2 Execution Environment

Model runs were performed on a variety of machines, from single processor PC's to Saguaro, a high performance computing cluster with 4560 processor cores [202]; all computers were running FLOSS operating systems, namely Linux or FreeBSD.

Parallelizing the code to take advantage of multiple processors was done as follows: FLOSSSim itself is single threaded. However, the nature of the genetic algorithm means that all individuals in a generation can be evaluated independently of one another. This makes it possible if the population size is N to use N processors to evaluate the population in $\frac{1}{N}$ the time, essentially resulting in a linear speed up limited by the size of the population. Therefore, when performing evolutionary runs, the number of threads was adjusted to take advantage of the number of processors available, significantly cutting down on wall clock time when exploring the model.

---

[4]http://cs.gmu.edu/ eclab/projects/ecj/

[5]The fact that ECJ is under an open source license allowed the library to be extended to include features needed for this research, such as modifications that allowed for reproducible results when running an arbitrary number of evaluation threads.

The code was profiled and manually optimized to decrease execution time. On average, a single 250 step run of the model, including collecting the statistics at the end of the run, took 0.78 seconds when executed on a computer with a 1.8 GHz Intel single-core processor with 1 GB of memory running Linux 2.6.21 and executing under Sun Microsystem's JAVA Development Kit 1.5.0_13. Each model parameter set was evaluated 8 times and the results averaged. Evolutionary runs consisted of 30 individuals per generation and 4096 generations, resulting in $8 \times 30 \times 4096 = 983040$ model runs per evolutionary run, or approximately 9 days of compute time. To reduce execution time, the data was scaled down for the simulation runs, using thousands of projects and agents rather than the hundreds of thousands found on SourceForge (see Section 7.2.5 for an analysis of running the model with a reduced number of projects and agents).

6.3.3 Verification

Traditional software engineering verification techniques were employed when developing the code to ensure correctness, including unit testing, integration testing, regression testing, and code reviews. In addition, FLOSSSim will be released on the OpenABM Consortium's website[6], a group which aims to improve the agent-based model development process through the FLOSS-like practice of encouraging researchers to release their model implementations so that others may independently verify, validate, and further study the models [203].

[6] http://www.OpenABM.org

CHAPTER 7

FLOSSSIM ANALYSIS

This chapter contains an analysis of FLOSSSim. After being calibrated and set up for testing as described in 6.2, the ability of FLOSSSim to reproduce and predict empirical FLOSS data is discussed in Section 7.1, along with an examination of the variance seen when using different success metrics. The sensitivity of the model to input parameters is presented in Section 7.2. Finally, using the model to explore scenarios is described in Section 7.3 and future work is discussed in Section 7.4.

7.1 RESULTS

The results of the model, after calibrating and using evolutionary computation to find well-performing sets of parameters, are included in the following sections. The model's ability to match the three empirical patterns is discussed in Section 7.1.1 and the ability to predict additional patterns, including distributions the model was not calibrated for, is shown in Section 7.1.2. An analysis of the parameters evolved by the genetic algorithm is contained in Section 7.1.3. Finally, the effects of using different success metrics in the model is examined in Section 7.1.4.

7.1.1 Matching Distributions

The averaged data over eight runs from the model's best parameter set (i.e., the parameter set resulting in the highest fitness score), along with the empirical data for the three patterns to be matched, are shown in Figs. 7.1, 7.2, and 7.3.

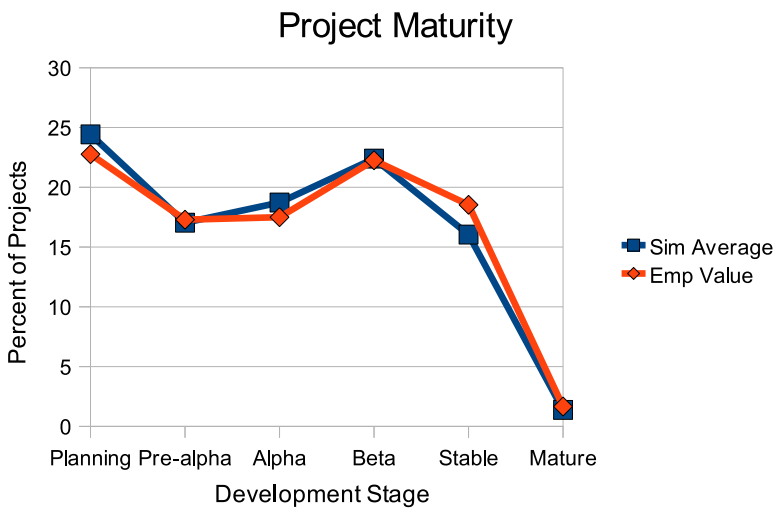


Fig. 7.1. Percentage of FLOSS projects in development stages. Empirical data from [117].

Figure 7.1 shows the model's percentage of projects in each development stage is very similar to the empirical data. During the development of the model, this has been the hardest distribution to match. In earlier versions of the model, new projects were all created in the planning stage [204]. In reality, this is not the case; many projects, by the time they are released as open source on SourceForge, are beyond the planning stage. In fact, only 26.1% of projects start in the planning stage when they first appear on SourceForge, as shown in Table 6.3. Mimicking this distribution when creating projects adds realism to the model and has improved the model's ability to match the empirical data.

Two thirds of FLOSS projects have only a single developer and 90% have fewer than four developers [117]. As shown in Fig. 7.2, the number of developers per projects follows a near-exponential distribution, and the simulated data is similar, especially for projects with fewer than 13 developers. Note that the data in Fig. 7.2 uses a logarithmic scale on the y-axis to help with a visual comparison between the two data sets. Beyond 13 developers, the values match less closely, although this difference is visually amplified as

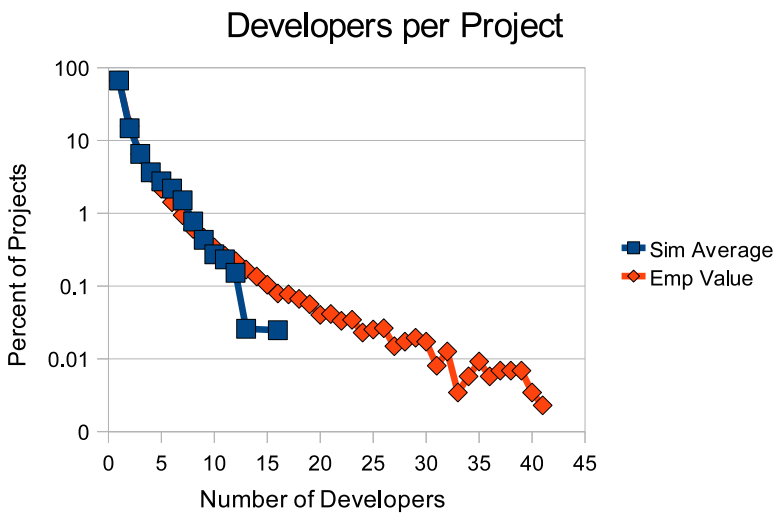


Fig. 7.2. Percentage of projects with N developers. Empirical data from [117].

a result of the logarithmic scale on Fig. 7.2 and is actually not as large as it might initially appear (e.g., for 15 developers, the difference between the empirical and simulated values is only 0.06%). Since there are very few projects with large numbers of developers in the empirical data, the higher values may be in the noise anyhow and thus focus should be on the similarity of the lower numbers.

In FLOSS, the distribution of projects per developer is highly skewed, with the majority of developers working on one or just a few projects [1], [77]. Meanwhile, a small subset of developers work on many projects [1], [80]. Figure 7.3 shows the model performs well in matching this distribution as well.

Table 7.1 contains the mean fitness scores for each of the emergent properties for the top performing parameter set. These values provide a quantitative mechanism for confirming the visual comparisons made above. Indeed, all three fitness scores are high, indicating good matches for all three distributions. The combined fitness is simply the mean of the three fitness scores, although this value could be calculated with uneven weights if, say,

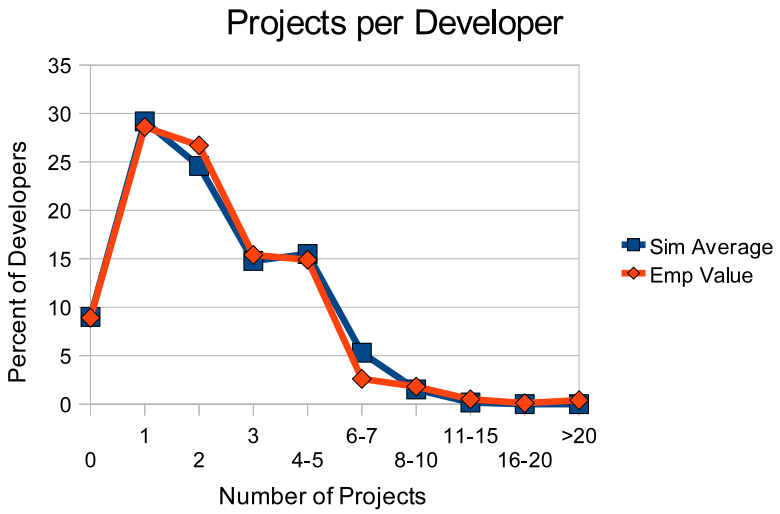


Fig. 7.3. Percentage of developers working on N projects. Empirical data from [1].

TABLE 7.1
Mean fitness scores of the best performing parameter set for the three emergent properties.

Emergent Property	Fitness Score
Maturity stage	0.99947
Developers per project	0.99988
Projects per developer	0.99934
Combined (mean)	0.99956

matching each property was prioritized. Doing so would affect how the genetic algorithm explored the parameter space. It may be the case that certain properties are easy to reproduce in the model and work over a wide range of parameter sets, in which case these properties may be weighted less than properties that are more difficult to match. Properties which are always matched should be discarded from the model for evolution purposes as they do not discriminate against different parameter sets.

Analysis of the model runs show that none of the three distributions are trivial to match. To demonstrate this, 1000 random parameter sets were evaluated, representing a random set of points in the state space landscape. If a distribution is trivial to match, it

TABLE 7.2
Range of fitness scores and percent of scores exceeding 0.98 for a random sample of 1000 parameter sets.

	Fitness Score Range	Fitness Scores ≥ 0.98
Maturity stage	$[0.4184, 0.9996]$	45.9%
Devs per project	$[0.7419, 0.9996]$	18.4%
Projects per dev	$[0.5576, 0.9967]$	3.0%
Combined (mean)	$[0.6532, 0.9904]$	0.9%

will have high fitness scores over most or all of the state space; that is, it will be relatively non-discriminating and thus non-helpful in locating near-optima in the landscape. Through experience gained while developing the model, a fitness value of 0.98 is chosen as the threshold to distinguish between good and bad fit[1]. The range of values and the percent of fitness scores exceeding 0.98 are shown in Table 7.2. The wide ranges of values demonstrate that the fitness values are not high over the entire state space; indeed, many parameters sets result in very poor matches for each of the distributions. The maturity stage fitness is greater than 0.98 for almost 46% of the random sample, indicating this may be the least discriminating of the three distributions. Although this value is high, the majority of the state space still results in poor fitness values, indicating matching this distribution helps calibrate the model. The developers per project and projects per developer have high fitness over a much smaller percent of the landscape, indicating these distributions are harder to match. The combined values exceed 0.98 for only 0.9% of the random parameter sets,

[1] A 0.98 fitness result is actually not a very good fit between the simulated and empirical data – using a genetic algorithm it has been possible to consistently find fitness values well exceeding 0.99 – but for this analysis it is preferable to error on the side of categorizing too many parameter sets as acceptable.

indicating matching all three distributions at the same time is difficult. Based on this, it is concluded that all three distributions contribute in tuning the realism of the model.

7.1.2 Predictive Validity

In this section, the capability of FLOSSSim to predict components of the FLOSS development process is explored. First, the ability of the model to predict future distributions for data that was used in the calibration process is analyzed in Section 7.1.2.1. Secondly, the ability of the model to predict values for which it was not calibrated is shown in Section 7.1.2.2.

7.1.2.1 Project Development Stage and Developers per Project Distributions

The model was calibrated for project development stage and developers per project distributions based on data from [117], which was collected in December 2004. As previously mentioned in Section 6.2.1, using the FLOSSmole database [156] these distributions were recalculated based on a June 2009 crawl of SourceForge. To determine how well the model performs in predicting these values, the model is first calibrated using the data from December 2004 and then run an extra 4.5 years, at which point the model's predicted values are compared to the June 2009 data. More specifically, the model is calibrated at 250 time steps using the 2004 data. The top performing 1% of parameter sets are then rerun, this time for 484 time steps, representing the original 250 time steps plus an additional 4.5 years of simulated time. The model's predicted results after 484 steps are then compared to the 2009 empirical data.

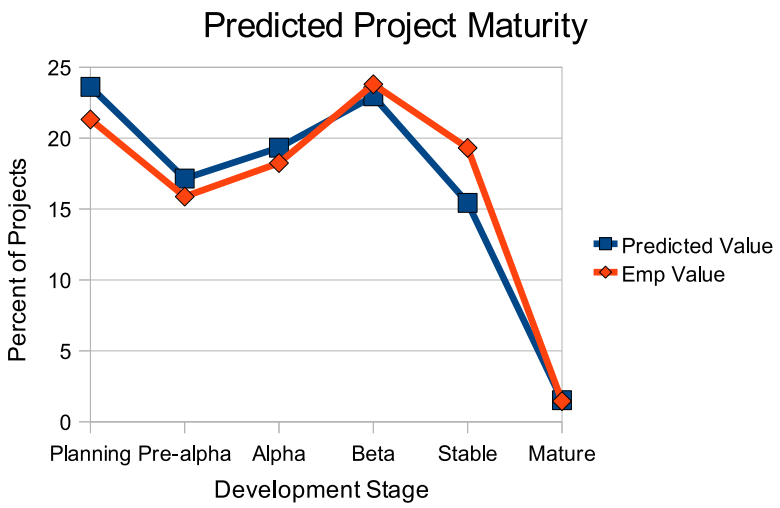


Fig. 7.4. Predicted versus empirical percentage of FLOSS projects in development stages in June 2009. The model is first calibrated with December 2004 data from [117] and then run for an additional 4.5 years. FLOSSSim's predictions closely match the empirical data. Empirical data compiled from FLOSSmole's [156] June 2009 crawl of SourceForge.

FLOSSSim's prediction for the maturity stage distribution is very similar to the 2009 SourceForge data. A sample run from one of the top performing parameter sets[2] is shown in Fig. 7.4. FLOSSSim predicts slightly too high values for the percentage of projects in the lower three development stages. The largest disparity between predicted and empirical values occurs in the production/stable stage, where FLOSSSim's estimate is low by less than 4%. Overall, the model performs well in predicting the project maturity stage distribution 4.5 years into the future.

A sample of FLOSSSim's prediction for the developers per project distribution is shown in Fig. 7.5. As can be seen, the model performs well in matching this distribution

[2]Results were relatively consistent across the top performing parameter sets and therefore only a sample is shown.

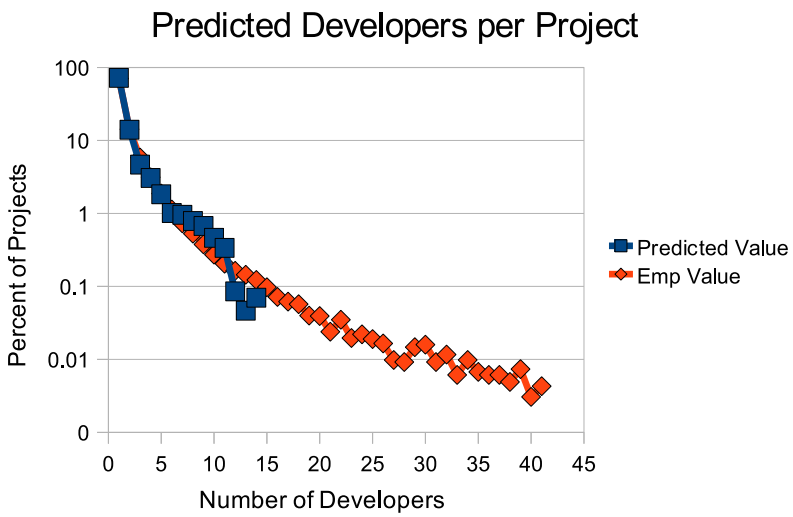


Fig. 7.5. Predicted versus empirical percentage of developers per project in June 2009. The model is first calibrated with December 2004 data from [117] and then run for an additional 4.5 years. FLOSSSim's predictions closely match the empirical data. Empirical data compiled from FLOSSmole's [156] June 2009 crawl of SourceForge.

as well for the bulk of projects. Only the extreme projects with greater than 14 developers, which account for only 0.63% of the projects, are not reproduced.

The ability of the model to accurate predict data 4.5 years into the future increases confidence that the model is realistic. This in turn adds confidence that the model may be used to explore and/or predict other aspects of the FLOSS development process, including components that cannot be directly validated.

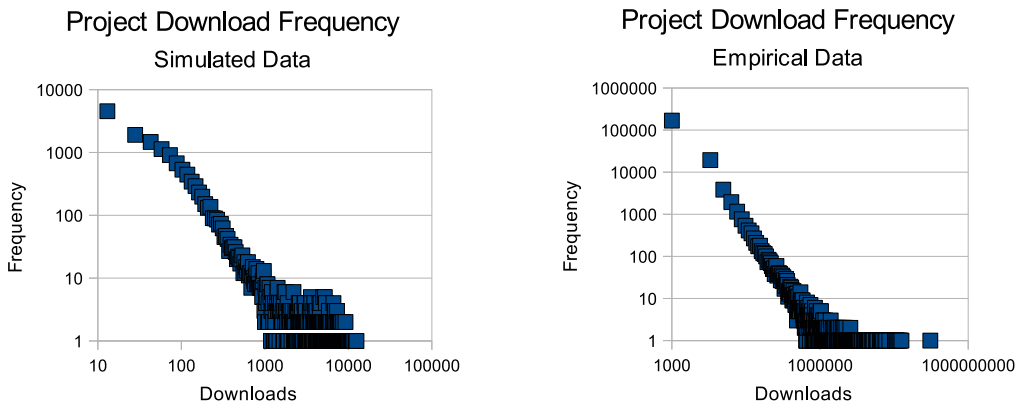
7.1.2.2 Downloads Distribution

Section 7.1.1 demonstrates that FLOSSSim is able to reproduce the three key patterns for which it was calibrated and Section 7.1.2.1 shows the model can predict future values for the calibrated data, but pattern-oriented modeling suggests that the structural realism imposed by matching multiple patterns may allow a model to match other patterns for which it was not originally designed [194]. To determine if the model performs well outside of the

patterns for which it was calibrated, a fourth pattern of downloads distribution is selected. The model's predicted values are then compared to empirical data.

The number of downloads SourceForge projects have received was mined from the FLOSSmole database. Prior to April 2005 inclusive, monthly download counts were stored in the database, while after April 2005 daily summaries were collected. Thus calculating the number of downloads involved joining data from multiple parts of the database. Because FLOSSmole crawls SourceForge roughly every 60 days, care had to be taken to avoid duplicate entries for a single day that occurred when two crawls were separated by fewer than 60 days. Records that included only a partial day's downloads were filtered out; in many cases, the complete day's download count was available from the subsequent crawl. Unfortunately, sometimes crawls occurred less frequently than every 60 days, causing holes in the daily download counts. In addition, as outlined in Section 4.2.2.3 on page 129, download counts from the database are expected to underrepresent the actual number downloads since software may be obtained via methods other than downloading from a project's SourceForge web page. As such, the calculated download counts should not be considered exact values, but because problems with the data affected all projects, the mined data is considered good enough to compare download distributions at a high level; specifically, underreported download counts for all projects is not expected to affect the shape of the distribution. The download counts were collected for 199,555 projects from the period of November 1999 through April 2009.

The frequency of downloads is shown in Fig. 7.6, with FLOSSSim's prediction in Fig. 7.6a and the empirical data in Fig. 7.6b. The shape of the distributions of the simulated and empirical data are similar. Both follow roughly a power-law distribution, appearing close to linear on a log-log scale when considering a line drawn through the average of



(a) Predicted downloads distribution. (b) Empirical downloads distribution.

Fig. 7.6. Predicted versus empirical downloads distribution. The shape of the two distributions is similar, indicating the model tracks reasonably with downloads even though it was not calibrated to predict these values.

the points. Both deviate slightly from this linearity for the number of projects receiving small numbers of downloads. As the frequency decreases, the range of downloads becomes more widely spread for both data sets. This makes sense, as failed projects all approach zero downloads, but the number of downloads successful projects receive is sensitive to other factors such as the size of the target audience, the elapsed time since the project was created, etc., hence the larger spread.

While the actual download counts and frequencies do not match between the empirical versus simulated data, for the purpose of this comparison the shape of the two distributions is more important. Having the actual values match is likely a matter of further calibrating the model and scaling the data. For example, to reduce execution time the model is run with fewer projects and agents than exist on SourceForge. Thus, the x-axis may be stretched or compressed by adding or removing consumer agents respectively to the model. Likewise, the y-axis may be adjusted by changing the number of projects in the simulation. Once the correct ratio of consumers to projects is determined, appropriate scaling factors

can be applied so that the model's output can be directly compared to the SourceForge data. Note that the three patterns used to calibrate the model are closely tied to producers and projects, but not consumers, and thus the calibration patterns likely provide little guidance in regards to determining consumer parameters. Therefore it is expected that additional calibration for the number of consumers is necessary to match consumer-based patterns, such as the project downloads distribution. The fact that the basic shapes of the two data sets are similar without explicit calibration is encouraging; it indicates that the model is structurally realistic enough to be used for predictive purposes and suggests that the fit may be further improved if calibration of this consumer component is performed.

In conclusion, although calibrated using three patterns, the model is able to closely match a fourth pattern, capturing the fact that most projects are infrequently downloaded while a few projects receive many downloads. It is worth noting that while the three calibration patterns are all developer-based, the model shows promise in predicting the fourth pattern which is user-based. This supports the notion that the model is structurally realistic, has predictive validity, and in general increases confidence in the validity of the model. Finally, with further calibration it is expected that the model may be able to accurately predict downloads.

7.1.3 Evolved Parameters

Some model parameters could not be estimated from empirical data. Instead, these parameters were evolved via a genetic algorithm to find values that allowed the model to closely match the empirical data. This section analyzes the evolved parameters that performed well in order to gain insight into the FLOSS development process.

7.1.3.1 Utility Weights

Examining the evolved utility weights of the best performing 1% of parameter sets (i.e., the sets resulting in the highest combined fitness values) provides insight into what factors are important in the model for reproducing the three emergent properties examined and consequently what factors are important to agents when selecting projects. Using normal probability plots, it was determined that the weights from the top 1% of parameter sets were not necessarily normally distributed. This may indicate that there are several good solutions to the problem located in disparate sections of parameter space. To discover groups of similar parameters, clustering is performed on the weights in the parameter sets. To accomplish this, each parameter sets' weights are placed in a vector of the form $(w_1, w_2, w_3, w_4, w_5)$. The k-means++ clustering algorithm [205] is used because it has been shown to reasonably address the problem of seeding the k-means clustering algorithm [206], which is sensitive to the selection of the initial centroids that, if chosen badly, may result in poor clusters. To further increase the probability of finding good clusters, the k-means++ algorithm is run 128 times and the run producing the tightest clusters is chosen[3]. For the purpose of forming clusters, the distance between two points is defined as the Euclidean distance.

To determine the correct number of clusters k, the clustering algorithm is run for values $k = 1$ to the number of data points. For each of these an error is calculated, defined as the sum of the square of the distances from each point to the centroid of the cluster to which the point belongs. The elbow method [207] is then used to determine the correct number

[3]Neither the k-means nor k-means++ algorithm are guaranteed to find the optimal clusters, although the k-means++ algorithm improves the chances of finding a good solution. Therefore, running the algorithm multiple times increases the chances of finding the best, or at least a better, set of clusters.

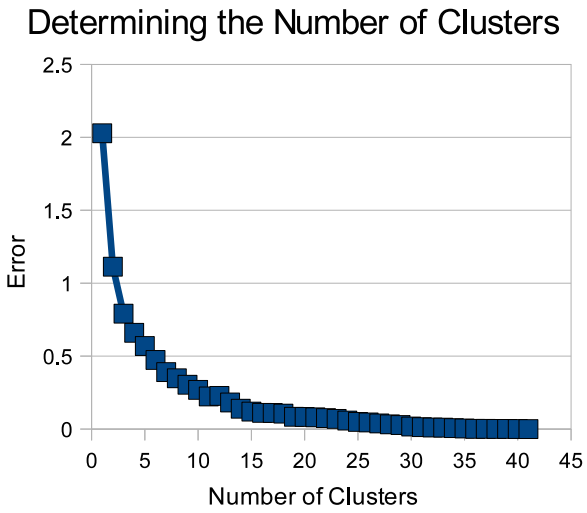
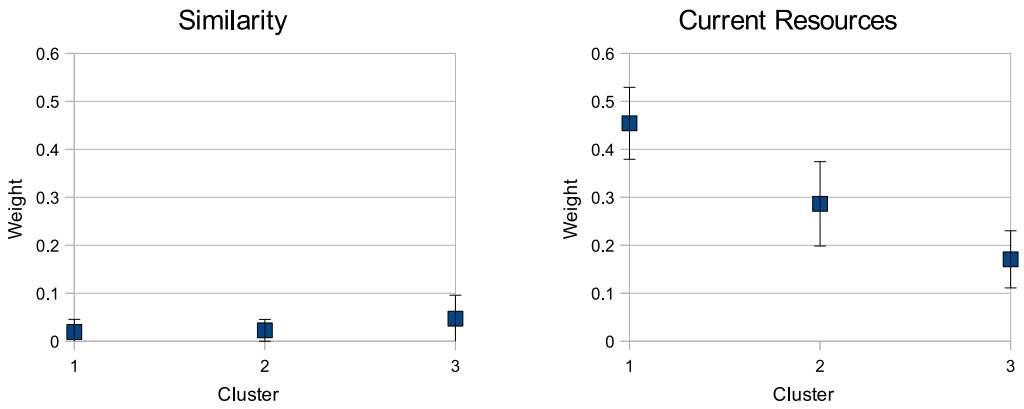


Fig. 7.7. Determining the number of cluster for the evolved utility weights. Error is defined as the sum of the square of the Euclidean distances between each point and its respective group's centroid. The data compresses well for three or fewer clusters, after which adding more clusters results in only a minor reduction in error. Therefore the correct number of clusters is three.

of clusters by plotting the number of clusters versus the error and finding the point where increasing k results in minimal additional compression as compared to previous increases.

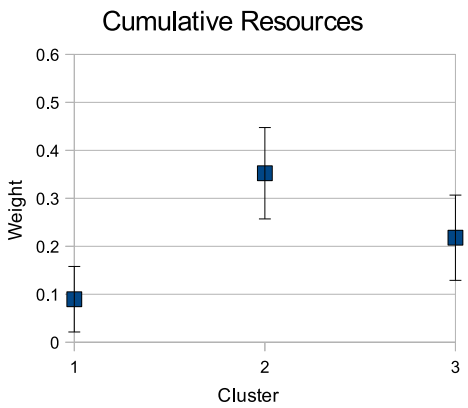
A plot of the number of clusters versus error is shown in Fig. 7.7. From this, it is concluded that the correct number of clusters is three. The resulting three clusters, grouped by weight, are shown in Fig. 7.8. Each point represents the mean weight and the error bars represent one standard deviation. The number of parameter sets in each cluster is contained in Table 7.3.

Across all clusters, the weight for similarity is stable and low, indicating similarity is not very important in matching the empirical data. The maturity stage weight is also stable across all three cluster, with its high value indicating the development stage of a project is always a significant factor in selecting projects. Finally, w_2, w_3, and w_4 vary across clusters, with each taking on a high value in a separate cluster. Specifically, w_2 is the most important

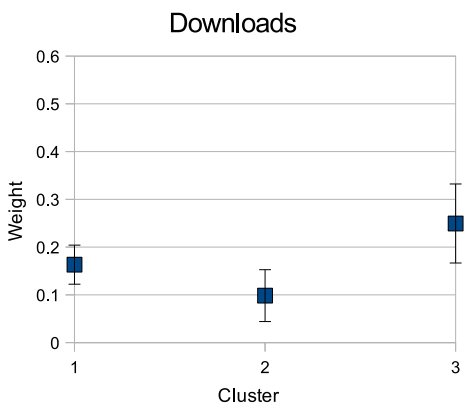


(a) w_1 utility weight clusters.

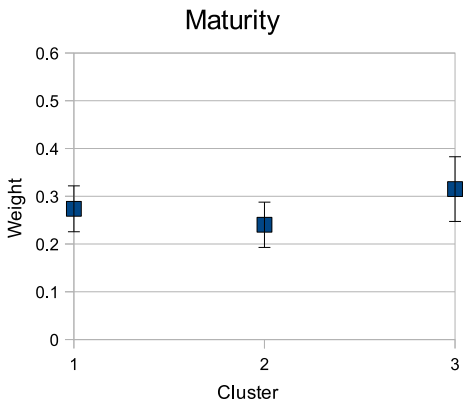
(b) w_2 utility weight clusters.



(c) w_3 utility weight clusters.



(d) w_4 utility weight clusters.



(e) w_5 utility weight clusters.

Fig. 7.8. Evolved utility weight clusters.

TABLE 7.3
Utility weight cluster sizes.

Cluster	Size	(Percent)
1	17	(41%)
2	8	(20%)
3	16	(39%)

factor in cluster 1, w_3 in cluster 2, and w_4 is high and the second most important factor in cluster 3. This indicates that there are three distinct agent profiles that work well to match the empirical data. These profiles are: agents that focus on the popularity of a project with developers (cluster 1), agents that key off the size and amount of work already completed on a project (cluster 2), and agents that consider the popularity of a project with users (cluster 3). The fact that there are multiple profiles that perform well is expected. Indeed, research has found heterogeneity in the factors that motivate developers to contribute to FLOSS; these motivating factors are also likely tied to how developers choose projects once they join the open source community. One survey [98] went so far as to cluster developers into four groups, namely believers, skill improvers, fun seekers, and professionals, based on the developers' motivation for being involved in open source.

An analysis of each evolved weight is contained in the following sections.

7.1.3.1.1 w_1 Similarity: w_1, the weight for the similarity between an agent and a project, is a stable value across groups, as shown by the close mean values and overlapping error bars seen in Fig. 7.8a. Surprisingly, the mean is rather low, averaging less than 5% in all cases, indicating the interests of an individual may not be one of the most important factors in selecting a project. A possible explanation for this comes from Linus' Law [13], which states that "Given enough eyeballs, all bugs are shallow." Similarly, with a large enough pool of heterogeneous developers, if a single developer finds a program

interesting enough to create as an open source project, then there will be other developers that also find the program interesting. This results in similarity being less of a driving force because there is always a pool of individuals interested in any project, and thus it is not a discriminating factor in matching the empirical data.

7.1.3.1.2 w_2 Current Resources: Current resources is a measurement of the development activity, and cluster 1 corresponds to a profile where the current resources is by far the most important factor in selecting a project, as seen in Fig. 7.8b. The importance of selecting projects that are being actively developed is a recurring theme in FLOSS literature. For example, reputation is cited as a reason developers become involved in open source development [1], [98]. Developers looking to boost their reputation should seek out active projects, as there is more potential for reputation gain in active projects than abandoned projects. Similarly, some developers become involved with open source to increase their job opportunities [1]. Active projects in effect provide better advertising for these developers because there is an increased chance that someone else working on the project will notice a developer's skills and approach him/her with a job opportunity. Sharing skills with others and learning new skills are two more reasons developers list for being involved in FLOSS, both of which rate high in developer surveys [1], [98]. Active projects provide a better opportunity for developers to show skills and/or pick up new skills from other talented developers. Based on this, a profile where agents highly value the development activity when selecting projects is expected.

7.1.3.1.3 w_3 Cumulative Resources: Cumulative resources is a measure of the size of a project completed so far, and cluster 2 corresponds to a profile where cumulative resources is the dominant factor when selecting a project, as seen in Fig. 7.8c. Developers may take into account the size of the project completed so far, as it may be more

difficult to break into the circle of developers in a small project than a large project, especially if the group of developers is already well-established and sufficient for the project. Developers interested in picking up new skills may gravitate towards larger projects as there may be more opportunities to gain knowledge when a large code base already exists.

7.1.3.1.4 w_4 Downloads: The number of downloads may be used as a proxy for the popularity of a project with users, and cluster 3 corresponds to an agent profile where downloads is an important factor when selecting projects, as seen in Fig. 7.8d. Some studies have shown that recognition is important and can be a driving force for those participating in online communities [138]. In some cases, individuals are willing to forego financial payment in exchange for recognition [136], [137]. Finally, it has been argued that some participate in open source because they are altruistic [13]. Cluster 3 supports these arguments, as there is more potential for recognition and a larger return on investment for contributions to projects that have many users.

7.1.3.1.5 w_5 Maturity: w_5, the maturity weight, is also a stable value for the top performing parameter sets. The means across all three clusters are similar and the variances are small and overlap, as can be seen in Fig. 7.8e. The lower standard deviation adds confidence that approximately 30% of a project's perceived utility is based on the development stage of the project. The high value for w_5, combined with the importance assigned to each development stage in (6.4), supports the hypothesis that developers prefer projects in earlier stages. It has been suggested this is because more reputation can be gained from developing core code. In addition, if developers are driven to join projects in order to resolve their own personal problems, as a project reaches the upper stages it transitions from development to maintenance, meaning there are fewer and fewer tasks remaining to be written that might solve a developer's problem and thus be an incentive to join the project.

TABLE 7.4
Evolved producer/consumer number distributions parameters.

Producer/Consumer Number		Parameter statistics from the top 1% of parameter sets	
		Mean	Stdev
Producer number	Mean	0.9285	0.0073
	Stdev	0.0613	0.0138
Consumer number	Mean	0.5609	0.2843
	Stdev	0.5827	0.2945

While empirical data supports the idea that projects with at least an initial working version of the software have increased chances of progressing further as open source projects (see Table 3.2), this information is not reflected in (6.4). However, this characteristic may still be captured in w_3, meaning the development stage is not as important as having at least skeleton code, reflected in the size of a project, that helps define the design such that the work can be divided among volunteers in an open source environment.

7.1.3.2 Producer and Consumer Numbers

Another interesting set of values evolved by the genetic algorithm are the parameters for the producer and consumer numbers. While the producer and consumer numbers are drawn from normal distributions bounded by 0.0 and 1.0 inclusive, neither the means nor standard deviations of these distributions are known. Therefore, these values are evolved to find the best performing values. The evolved mean and standard deviation for the producer and consumer numbers averaged from the top 1% of parameter sets are contained in Table 7.4. Notice that the mean producer number is very high at 0.9285 and relatively stable across the top 1% of parameter sets, with a standard deviation of 0.0073. Likewise, the producer number standard deviation is low at 0.0613 and also relatively stable with a standard deviation of 0.0138. This indicates that the top performing model runs have agents with high

propensities to develop. In other words, having most agents produce frequently (i.e., most agents be developers) results in matching the empirical data. This is in alignment with the notion that FLOSS is a developer-driven process [13], [66], [94], [101]. Furthermore, the evolved producer mean indicates that agents, on average, will be developing just under 93% of the time. This value is very similar to the open source developers surveyed in [1], where 91% reported being involved in one or more projects at the time the survey was conducted, and only 9% considered themselves FLOSS developers but were currently between projects. The similarity between these two numbers adds confidence about the validity of the model and the process employed to find correct model parameters through the use of genetic algorithms.

In contrast, the evolved consumer number mean is much lower and the standard deviation is much higher compared to the producer number mean and standard deviation respectively. Furthermore, neither the consumer number mean nor standard deviation is particularly stable, i.e., both have large standard deviations over the top performing parameter sets. This indicates that the consumer number distribution has little effect on matching the empirical data because values from 0.0 to 1.0 all appear to work equally well. In other words, consumers have minimal effect on the ability of the model to match the empirical data.

In conclusion, the best performing parameter sets all include many agents developing frequently, indicating developers are key to the model matching the empirical patterns. On the other hand, the frequency of agents consuming results in high fitness over a very wide range, indicating that while consumers may have some minor influence in the model, they are not the main driving force in matching the empirical data.

7.1.3.3 Maximum Number of Projects Producing and Consuming

A final set of values evolved by the genetic algorithm that should be examined are the upper limits for the number of projects an agent can produce and, separately, consume at any given time. Setting each of these maximums to appropriate values helps the model match the empirical patterns.

The evolved maximum number of projects an agent can produce is found to be 12 for all but one of the top 1% of the best performing parameter sets, in which case it is 11, making this value very stable across all runs. A sanity check of this evolved parameter can be performed by comparing it to empirical data from [1], seen in Fig. 7.3, which shows the percentage of developers working on N projects approaching zero for 11–15 projects. Although the average evolved maximum value precludes the model from ever matching the non-zero values of the empirical data above 12 projects, only 1% of developers fall into categories developing for more than 15 projects [1], making these developers unusual and arguably outliers. The fact that the model performs well when this value is calibrated adds confidence that the exponential distribution used in the model to control how many projects developers are contributing to is a decent approximation of the real distribution.

The evolved value for the maximum number of projects agents are able to consume is three for all but two of the top 1% of parameter sets, in which case it is four. Unlike the value for producing, the model performs best when users are limited to downloading only a small number of projects at a time. Note that three is at the bottom end of the range used for the evolutionary run (see Table 6.5 on page 182). Since the number of projects agents are able to consume approaches the minimum, this makes one question if consumers are actually hindering the model's ability to match empirical data. Maximum values below

three were not considered, as limiting consumers to using only one or two projects seemed unrealistic. The effects of consumers are further explored in Section 7.3.1.

Taken along with the evolved producer and consumer numbers, it has been shown that the empirical data is best matched when most agents are developers and some developers work on many projects. This further supports that developers are the driving force of the FLOSS development process and instrumental in allowing FLOSSSim to match the empirical data. On the other hand, the model is largely unaffected by the frequency of agents consuming and performs best when agents download only one or a few projects at a time. This suggests that users may not be an important component of causing the model to match the empirical data.

7.1.4 Success Metrics

A goal of this research is to better understand both what it means for a project to be successful and what conditions change the probability of success. To explore these concepts, FLOSSSim is used to compare the similarities and differences among a subset of success metrics in Section 7.1.4.1. The effects of popularity and target audience size on success are explored in Section 7.1.4.2.

7.1.4.1 Comparing Success Metrics

Although many success metrics have been proposed for FLOSS, it is unclear which are best or if there is even a significant difference among these metric. If there is a high correlation between certain metrics then the "easiest" metric, such as the metric that is most convenient to measure or collect, may be chosen over other metrics without affecting the results. To explore the similarities and differences, the following adaptations of proposed success metrics are considered:

Maturity threshold: Projects that are in the beta stage or higher are considered successful. This is based on the notion that projects that have reached the beta stage have produced useful software.

Δ maturity: In order to demonstrate satisfactory progress, projects must progress to a higher development stage every six months to be considered successful. Projects that are in the production/stable or mature stages are also considered successful, as these projects are at the upper limit and may remain active without ever moving stages.

Δ developers: The number of developers must increase every six months in order to demonstrate the vitality of the project.

Δ downloads: The number of downloads that occur in a six month interval must increase every six months in order to demonstrate user interest in the project.

Δ percent complete: At least 10% of the project must be completed every six months in order for the project to be considered making satisfactory progress.

Completed projects: Projects that are complete are considered successful.

Some of these metrics are inherently difficult to measure for real projects, especially considering the amount of dirty data from sites like SourceForge or included in the FLOSSmole database. Therefore, the model is used to draw inferences about the real data. These particular success metric were chosen because they mapped into the model and therefore were feasible to collect. The six month time limit used in several of the metrics was chosen as a reasonable limit based on existing literature. As per [13]'s recommendation, open source projects should release early and release often in order to be successful. [72] uses a similar timeframe to the one chosen here, classifying projects as successes or failures based

TABLE 7.5
Percentage of successful projects based on different success metrics.

Success Metric	Successful Projects
Maturity threshold	39.83%
Δ maturity	18.02%
Δ developers	19.17%
Δ downloads	9.93%
Δ percent complete	0.01%
Completed projects	1.41%

on if a version is released within one year. Similarly, [48] believes successful projects will release a stable version within six months and average at least one release per year. Therefore, the six month timeframe in the above metrics seems to be in a range that is generally accepted by other FLOSS researchers.

Using the evolved parameter set that produced the highest fitness, the percentage of projects at the end of a run meeting each of the selected success criteria was calculated. The results are shown in Table 7.5. From these results, it is clear that there are significant differences among the varying success metrics. The least discriminating metric, maturity threshold, categorizes almost 40% of the projects as successful while the Δ percent complete metric considers almost none of the projects successful. What is not clear is how much overlap there is among the different metrics. Are most of the projects categorized as successful by one metric also considered successful by other metrics? To consider the overlap among sets of successful projects, the Jaccard similarity is calculated between each pair of metrics. The similarity values between sets are surprisingly small, indicating there is minimal overlap across the various metrics, as shown in Table 7.6. This indicates that the proper choice of metric(s) is important as the metrics themselves differ. That is, "successful" projects may not be successful according to all metrics. It has been suggested that

TABLE 7.6
Jaccard similarity between different sets of successful projects.

	Maturity threshold	Δ maturity	Δ developers	Δ downloads	Δ percent complete	Completed projects
Maturity threshold	1.00	0.44	0.15	0.10	0.00	0.04
Δ maturity	0.44	1.00	0.1	0.09	0.00	0.08
Δ developers	0.15	0.10	1.00	0.18	0.00	0.00
Δ downloads	0.10	0.09	0.18	1.00	0.00	0.03
Δ percent complete	0.00	0.00	0.00	0.00	Undef.	0.00
Completed projects	0.04	0.08	0.00	0.03	0.00	1.00

multiple metrics be used in calculating if a project is successful [70], as this may result in a more balanced evaluation process.

7.1.4.2 Target Audience Size versus Success

In order for projects to be successful, they must manage to attract and maintain developers and, possibly, users. The similarity weights evolved by FLOSSSim are surprisingly low (see Section 7.1.3.1), indicating that matching agents' interests with projects has only a small influence when selecting a project. Yet there is a common theme in FLOSS literature that the target audience of a project does affect a project's prospects of being successful. For example, [66] finds evidence that projects with topics aimed at certain target audiences are more successful than projects aimed at other target audiences. With this in mind, one might expect that the projects with the largest target audiences are also the most popular, and the most popular projects, by being able to tap into the skills of a larger segment of the developer and consumer pool, are also more likely to be successful. FLOSSSim is used to further explore this idea.

The notion of largest target audience can be captured in FLOSSSim by measuring the distance between an agent's and project's needs vectors. Essentially, the size of the target audience can be determined by counting the number of nearby agents surrounding a project. If each project is categorized as successful or not successful, statistical analyses can then be performed to determine if there is a relationship between the size of the target audience and the success of a project.

To determine if the number of agents near a project influences the project's success, FLOSSSim is run multiple times, each time configured with parameters from the best performing evolved parameter sets. For each run, at the end of 250 time steps the number of agents nearby each project is counted. In order to count the number of agents that are considered near a project, a distance threshold must be chosen. This was determined experimentally and ultimately set to 0.15, which allowed for a sufficient spread such that there was variety in the number of agents near projects but at the same time the distribution wasn't too spread out. A histogram showing the distribution from all runs can be seen in Fig 7.9. Note that agents may be counted multiple times if they fall within the distance diameter of multiple projects. In addition to counting the number of nearby agents, at the completion of each run each project is tested to see if it is successful according to each of the six success metrics outlined in Section 7.1.4.1. The results of all runs are combined and then binary logistic regression is used to determine if the number of nearby agents has an effect on the success of a project.

The results of the binary logistic regression for all six success metrics are shown in Figs. 7.10–7.15. Only the results from the Δ developers and Δ downloads success metrics have P-values < 0.05, indicating there is sufficient evidence for these two metrics that the model coefficients are non-zero using an α-level of 0.05. However, for the Δ developers

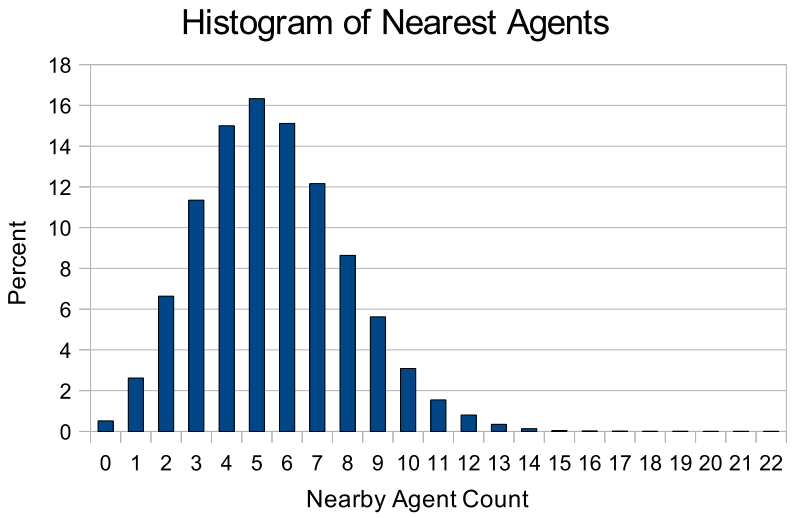


Fig. 7.9. Distribution of nearby agent counts when the distance threshold is 0.15.

metric the goodness-of-fit tests' P-values are < 0.05, indicating the null hypothesis that the model adequately describes the data should be rejected. This means there is sufficient evidence that the number of nearby agents is in fact linked to the success of a project only when measuring success based on Δ downloads. In this case, the coefficient is a small positive number and the odds are only slightly greater than 1.0, indicating an increase in the number of nearby agents will only slightly increase the chances that a project will be successful. Note that this connection is logical, as there is a direct link between number of nearby agents and the number of downloads a project receives. That is, the number of downloads a project receives is influenced by the number of nearby agents that choose to consume. This should remain true so long as w_1, the similarity weight in the utility function, is non-zero. The fact that w_1 is small in the top performing parameter sets is reflected in the weak link found in the binary logistic regression. While similarity plays a role in determining the utility of a project, it only plays a small role. Thus, an increase in nearby agents only results in a small increase in the chances that more agents will use a project. One might expect to see

```
Binary Logistic Regression: Maturity Threshold versus Nearby Agent Count

Link Function: Logit


Response Information

Variable            Value  Count
Maturity Threshold  1      28803  (Event)
                    0      43965
                    Total  72768


Logistic Regression Table

                                                          Odds     95% CI
Predictor             Coef    SE Coef       Z      P  Ratio  Lower  Upper
Constant        -0.430200  0.0187632  -22.93  0.000
Nearest agents  0.0013184  0.0031057    0.42  0.671   1.00   1.00   1.01


Log-Likelihood = -48847.628
Test that all slopes are zero: G = 0.180, DF = 1, P-Value = 0.671


Goodness-of-Fit Tests

Method           Chi-Square  DF      P
Pearson             16.6598  21  0.732
Deviance            18.9702  21  0.587
Hosmer-Lemeshow      2.8934   5  0.716
```

Fig. 7.10. Binary logistic regression for maturity threshold success metric versus number of nearby agents.

a similar link for the Δ developers metric because developers with interests near a project are also slightly more likely to contribute to the project than developers with dissimilar interests, but as already pointed out the binary logistic regression model is a poor fit for this data. The remaining four success metrics are a step removed from such direct links, e.g., Δ maturity may be influenced by the number of nearby developers, but it also depends on the amount of resources that are being contributed by those developers, the size of the project, etc.

```
Binary Logistic Regression: Change in Maturity versus Nearby Agent Count

Link Function: Logit


Response Information

Variable          Value  Count
Changed Maturity  1      12548  (Event)
                  0      60220
                  Total  72768


Logistic Regression Table

                                                          Odds     95% CI
Predictor            Coef    SE Coef       Z      P  Ratio  Lower  Upper
Constant         -1.60183  0.0243305  -65.84  0.000
Nearest agents  0.0060298  0.0040126    1.50  0.133   1.01   1.00   1.01


Log-Likelihood = -33452.627
Test that all slopes are zero: G = 2.255, DF = 1, P-Value = 0.133


Goodness-of-Fit Tests

Method           Chi-Square  DF      P
Pearson             11.1107  21  0.961
Deviance            13.2188  21  0.901
Hosmer-Lemeshow      2.6573   5  0.753
```

Fig. 7.11. Binary logistic regression for Δ maturity success metric versus number of nearby agents.

Note that in this experiment there is no attempt to differentiate between consumers and producers. All agents are treated equally when counting the number of agents near a project, even though some types of agents (e.g., developers) may have a larger influence on the success of a project than others types of agents (e.g., consumers).

In conclusion, the regression analyses indicate there is only a statistically significant correlation between the number of nearby agents and the success of a project for one of the six success metrics. For the Δ downloads metric the link is very weak, indicating that

```
Binary Logistic Regression: Change in Developers versus Nearby Agent
Count

Link Function: Logit


Response Information

Variable                   Value  Count
Change in Developer Count  1       9311  (Event)
                           0      63457
                           Total  72768


Logistic Regression Table

                                                          Odds     95% CI
Predictor             Coef    SE Coef       Z      P  Ratio  Lower  Upper
Constant         -2.11891  0.0278277  -76.14  0.000
Nearest agents  0.0356413  0.0044894    7.94  0.000   1.04   1.03   1.05


Log-Likelihood = -27801.035
Test that all slopes are zero: G = 62.524, DF = 1, P-Value = 0.000


Goodness-of-Fit Tests

Method           Chi-Square  DF      P
Pearson             46.5954  21  0.001
Deviance            52.8371  21  0.000
Hosmer-Lemeshow     26.5737   5  0.000
```

Fig. 7.12. Binary logistic regression for Δ developers success metric versus number of nearby agents.

the number of nearby agents has only a small influence on the success of a project. This is consistent with earlier results that indicate the similarity between an agent and a project is not a major driving factor in determining which projects are selected for contributions or download. This is somewhat surprising, as other studies have shown there are differences in project success based on target audience. However, the weak link discovered here may be more pronounced if, for example, the distribution of agents needs vectors around projects was further skewed. FLOSSSim currently distributes agents' needs vectors approximately

```
Binary Logistic Regression: Change in Downloads versus Nearby Agent
Count

Link Function: Logit


Response Information

Variable             Value  Count
Change in Downloads  1       4357  (Event)
                     0      68411
                     Total  72768


Logistic Regression Table

                                                          Odds     95% CI
Predictor            Coef    SE Coef       Z      P  Ratio  Lower  Upper
Constant         -2.98422  0.0393328  -75.87  0.000
Nearest agents  0.0409120  0.0062852    6.51  0.000   1.04   1.03   1.05


Log-Likelihood = -16470.046
Test that all slopes are zero: G = 41.866, DF = 1, P-Value = 0.000


Goodness-of-Fit Tests

Method           Chi-Square  DF      P
Pearson             19.1331  21  0.577
Deviance            23.1036  21  0.338
Hosmer-Lemeshow      2.7680   5  0.736
```

Fig. 7.13. Binary logistic regression for Δ downloads success metric versus number of nearby agents.

evenly around interest space as a result of using an uniform random number generator when generating needs vectors. If in the real world people's interests cluster around popular topics, then projects near these clusters of people's interests may perform better. This may result in more of the success metrics showing a statistically significant link (and possibly stronger link) between the number of nearby agents and the success of a project. Unfortunately, there is no method to accurately measure the real world interests of people involved in open source for the purpose of generating needs vectors more realistically, which is why

```
Binary Logistic Regression: Change in Percent Complete versus Nearby
Agent Count

Link Function: Logit


Response Information

Variable                    Value  Count
Change in Percent Complete  1         16  (Event)
                            0      72752
                            Total  72768


Logistic Regression Table

                                                          Odds     95% CI
Predictor            Coef   SE Coef       Z      P  Ratio  Lower  Upper
Constant         -7.87748  0.590383  -13.34  0.000
Nearest agents  -0.104241  0.108316   -0.96  0.336   0.90   0.73   1.11


Log-Likelihood = -150.277
Test that all slopes are zero: G = 0.962, DF = 1, P-Value = 0.327


Goodness-of-Fit Tests

Method           Chi-Square  DF      P
Pearson             9.59669  21  0.984
Deviance            9.11537  21  0.988
Hosmer-Lemeshow     8.51295   5  0.130
```

Fig. 7.14. Binary logistic regression for Δ percent complete success metric versus number of nearby agents.

a uniform distribution is used in FLOSSSim. However, if the percentage of existing FLOSS projects for various categories of software can be used as a proxy for people's interests, then there is some evidence that interests are not uniformly distributed. See [1] for an example distribution of the types of projects surveyed developers contribute to.

7.2 Sensitivity Analysis

Sensitivity analyses are performed by sweeping a single model parameter through a range of values while holding all other parameters constant. The sensitivity analysis of the model

```
Binary Logistic Regression: Completed Projects versus Nearby Agent Count

Link Function: Logit


Response Information

Variable            Value  Count
Completed Projects  1        114  (Event)
                    0      72654
                    Total  72768


Logistic Regression Table

                                                          Odds     95% CI
Predictor             Coef    SE Coef       Z      P  Ratio  Lower  Upper
Constant         -6.79271   0.238181  -28.52  0.000
Nearest agents  0.0588159  0.0372045    1.58  0.114   1.06   0.99   1.14


Log-Likelihood = -848.994
Test that all slopes are zero: G = 2.447, DF = 1, P-Value = 0.118


Goodness-of-Fit Tests

Method           Chi-Square  DF      P
Pearson              9.6824  21  0.983
Deviance            10.7002  21  0.968
Hosmer-Lemeshow      5.1764   5  0.395
```

Fig. 7.15. Binary logistic regression for completed projects success metric versus number of nearby agents.

serves several purposes. In some cases it allows for estimating parameter values which cannot be estimated via other methods. In these cases, the best working values from the parameter sweeps are chosen for the parameters. Where parameters are already estimated, sweeps of nearby values demonstrate how sensitive the model is to these values. This is especially important where only weak or approximate estimates of values may be available, as the sensitivity analysis provides feedback in regards to how critical these values are to

the performance of the model. Finally, understanding the ranges of values that work well for model parameters in turn provides insight into the FLOSS development process.

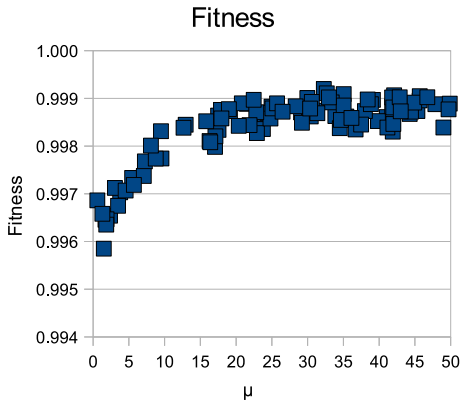
The following sections contain sensitivity analyses performed for model parameters.

7.2.1 μ

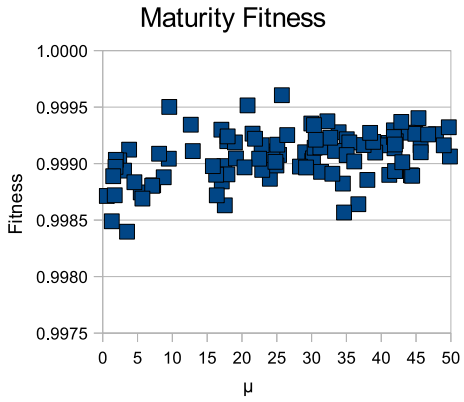
Agents making perfect choices when selecting projects is controlled by μ (see (6.5)). To determine the sensitivity of the model to the level of perfect choice, the following analysis is performed: All parameters besides μ are held constant, set to the values from the best performing evolved parameter set. 100 runs are performed, varying μ randomly in the range $[0.0, 50.0]$. The relationship between μ and the model's fitness score is then examined. The procedure is repeated multiple times with other parameter sets from the top 1% of evolved parameter sets to demonstrate consistency.

Scatterplots of μ versus fitness from a sample parameter set is shown in Fig. 7.16. The combined fitness is poor for low values of μ and improves as μ approaches 30, as shown in Fig. 7.16a. Above 30, the fitness values level off and higher values of μ do not appear to provide any significant improvement to fitness. To better understand where the changes in combined fitness originate, the relationships between μ and each of the three patterns used to calculate the combined fitness are shown in Figs. 7.16b–7.16d.

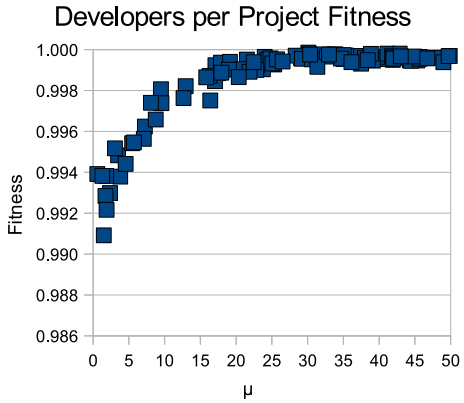
One might expect the maturity distribution fitness, shown in Fig. 7.16b, to change with μ. When μ is low, developers flit from one project to another, versus when μ is large, developers give their resources to a more focused group of projects. Thus as μ decreases, one might expect a shift from a few projects moving to the upper development stages, as a result of contributions from repeat developers, to a very slow increase in maturity of all



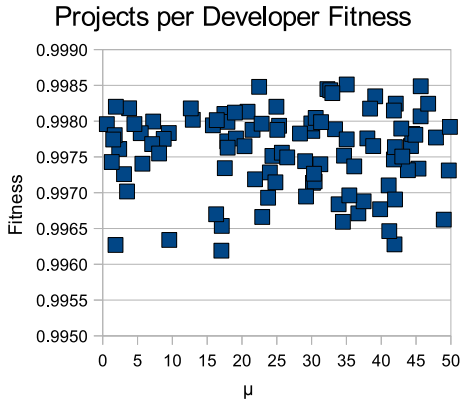
(a) μ versus combined fitness.



(b) μ versus maturity fitness.



(c) μ versus developers per project fitness.



(d) μ versus projects per developer fitness.

Fig. 7.16. Effects of varying μ on the fitness of the model. μ impacts the combined fitness of the model by affecting the model's ability to match the developers per project distribution. Values above 30 result in high fitness.

projects, as a result of developers spreading their contributions across all projects roughly equally. However, as already shown in Table 3.2 on page 102, projects rarely change development stages to begin with, so while the few projects that rapidly progress may be eliminated by reducing μ, this number is dwarfed by the number of projects that make minimal or no progress. Thus there is no significant change in the the maturity distribution, especially for runs of this length, over the range of μ. Longer runs may see a larger impact from changes to μ.

μ plays a significant role in causing the model to match the developers per project pattern, as shown in Fig. 7.16c. With lower values of μ, imperfect choice sets in and developers move towards choosing randomly from the projects in their memory. This behavior is incompatible with reproducing the developers per project distribution, where many projects have a small number of developers and a few have a large number of developers. By spreading the developers more evenly across the landscape of FLOSS projects, random selection reduces the chances of a project acquiring many developers. In the real world, many developers stay long-term with projects, which allows a small minority of projects to accumulate larger numbers of developers. Near perfect choice promotes this behavior by causing developers to contribute to the same projects multiple times; this in turn promotes matching the developers per project pattern. Note, however, that there is a diminishing return on increasing μ: beyond 30, the improvement is negligible. This is because a value of 30 results in near perfect choice already; that is most of the time an agent will pick the project with the highest utility and increasing μ beyond 30 results in only the occasional improvement in choice.

Finally, μ should have no effect on the projects per developer distribution, and Fig. 7.16d confirms it does not.

The performance of μ exhibited in this sensitivity analysis is consistent with the performance seen during model development. When using genetic algorithms to evolve μ to a value in the range $[0.0, 50.0]$, the best performing parameter sets always had evolved values for μ in the 30's and low 40's. This has been consistent across many variations of the model that were tested during development and indicates that μ may be an independent variable. 36 was chosen as a good static value for μ based on the average from a number

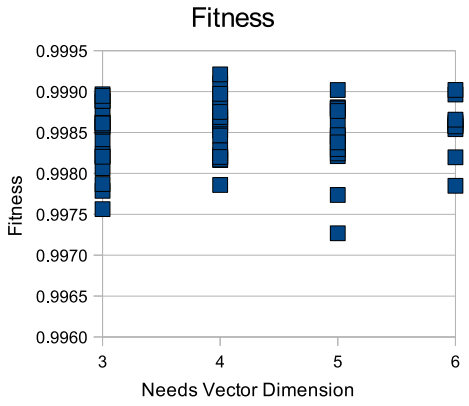
of experimental runs, putting it in the range of best performing values per the sensitivity analysis.

7.2.2 Needs Vector Dimension

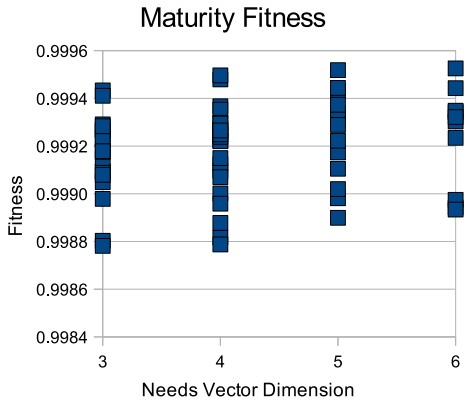
As mentioned in Section 6.2.2.2.2, when setting the dimensionality of the needs vector, the goal is to find the minimum value that results in sufficient variation in the model's agents and projects in order to reproduce the empirical data. To determine the effect of dimensionality on fitness, once again all parameters are held constant with the exception of the dimensionality, which is assigned values between three, the smallest value that is considered to provide a reasonable level of complexity when representing agents' interests, and six. The analysis was repeated multiple times with different parameter sets that were known to perform well. The results from a sample parameter set are shown in Fig. 7.17. There is no trend indicating that the overall fitness (Fig. 7.17a) nor the individual pattern fitness values (Figs. 7.17b–7.17d) perform better or worse as the dimensionality increases. This indicates that three is at or above the minimum value necessary to create sufficient variation in the model in order to match the patterns. Thus, the value of three is used in the model because it performs as well as higher values while requiring fewer calculations when computing similarity between agents and projects.

7.2.3 Starting Memory Size

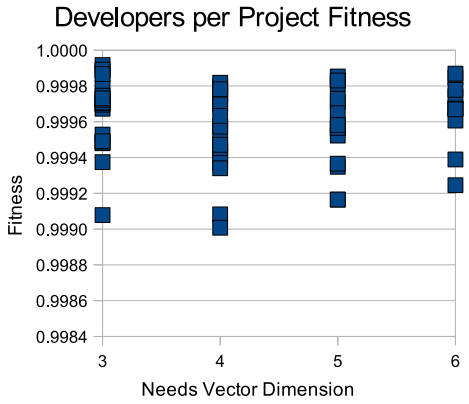
An analysis similar to the previous sections was performed to determine the optimum starting memory size. While all other parameters were held constant, runs were performed while varying the starting memory size in the range of $[5, 25]$. Plots of the resulting fitness values have been omitted for brevity's sake. The starting memory size has an effect on the



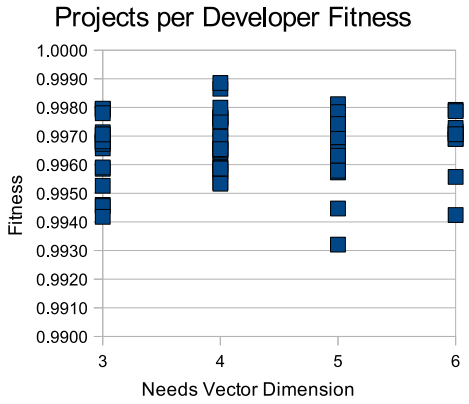
(a) Needs vector dimension versus combined fitness.



(b) Needs vector dimension versus maturity fitness.



(c) Needs vector dimension versus developers per project fitness.



(d) Needs vector dimension versus projects per developer fitness.

Fig. 7.17. Effects of varying the needs vector dimension on the fitness of the model. Higher dimensions do not increase fitness, indicating the minimum value of three provides sufficient variation in the model to match the patterns.

number of developers per project distribution, with the lowest values resulting in the highest fitness. This somewhat suprising result of the best fitness values being obtained when agents' memories are primed with only a few projects can be explained due to scaling in the model. To reduce execution time, FLOSSSim model runs are performed with scaled down data, including a significantly reduced number of projects compared to the number available in the real world. Therefore, the number of projects an agent is aware of is also scaled down. Based on the sensitivity analysis, the value of five was chosen for the model.

7.2.4 Memory Change Probability

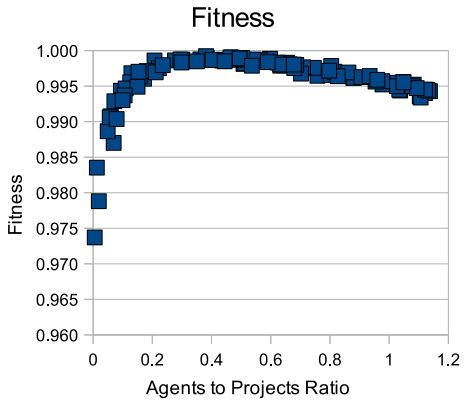
An analysis similar to the previous sections was performed to determine the optimum value for the memory change probability. The probability was varied in the range $[0.05, 0.5]$ while all other parameters were held constant. Plots of the resulting fitness values have been omitted to save space. Like the starting memory size, the memory change probability has an effect on the number of developers per project. When this value is high, agents' memory become volatile and change rapidly. This results in FLOSSSim not reproducing the long tail seen in the empirical data. This is because the more volatile the memory, the less likely it is that agents will work on a project long-term, meaning projects will not manage to accumulate many developers. The model exhibited the highest fitness when the memory change probability was 0.065. On average, this translates to just over six memory changes per year (i.e., three adds and three removes). This number seems reasonable when taking into account that the FLOSSSim runs include a scaled down number of projects.

7.2.5 Number of Agents and Projects

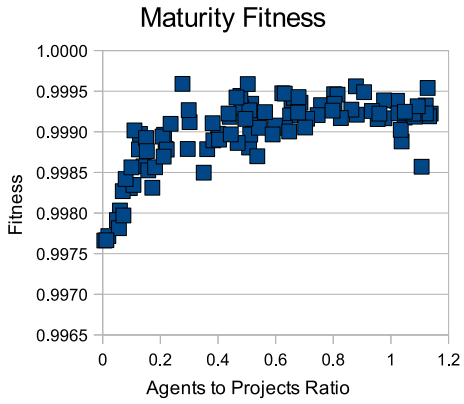
An analysis similar to the previous sections was performed to determine the optimum ratio of agents to projects. All parameters were held constant while varying the ratio of agents to projects in the range $[0, 16]$[4]. Scatterplots from an example parameter set showing the relationship between agents-to-projects ratio and fitness for ratios less than 1.2 are shown in Fig. 7.18.

As expected, the ratio of agents to projects has an effect on all distributions. The main effect is on the developers per project, which peaks when the ratio is between 0.3 and

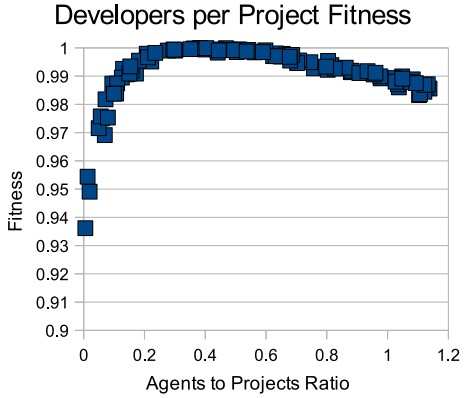
[4]The ratio was varied by holding the number of projects constant and only varying the number of agents.



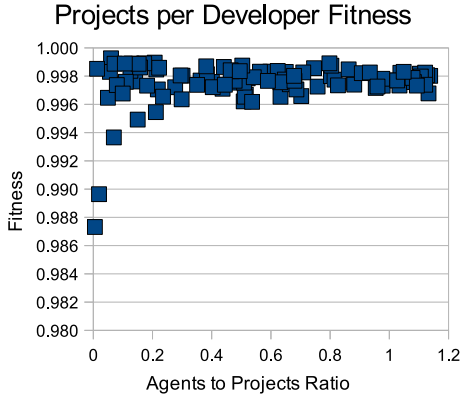
(a) Agent-to-project ratio versus combined fitness.



(b) Agent-to-project ratio versus maturity fitness.



(c) Agent-to-project ratio versus developers per project fitness.



(d) Agent-to-project ratio versus projects per developer fitness.

Fig. 7.18. Effects of varying the agents-to-projects ratio on the fitness of the model. Varying the ratio impacts all three fitness components, with the maximum fitness occurring at different ratio values for each of the components. In general, values below 0.2 perform poorly.

0.4, as seen in Fig. 7.18c. For a given ratio, there is minimal variability in fitness scores for this indicator. The fitness of the projects per developer distribution does not have a clearly defined maximum but seems to be high for ratios greater than 0.2, as shown in Fig. 7.18d. The peak maturity fitness occurs at different ratio values depending on the parameter set being evaluated. An example, where high ratios result in high maturity fitness values, is shown in Fig. 7.18b. For all three distributions there is a sharp decline in fitness when the

ratio is too low. Ratios that are too high exhibit much more gradual changes in fitness. 0.38 was determined to be the optimal ratio of agents to projects based on the combined fitness from a series of runs using multiple well-performing parameter sets.

To determine the absolute number of projects sufficient to replicate the data, evolutionary runs were performed with 1024, 2048, and 4096 initial projects while maintaining the ideal 0.38 ratio of agents to projects. Values less than 1024 were not tested because they are considered too small for the system being modeled. Regardless of the starting number of projects, all runs evolved similar parameters and had similar fitness values and the same number of clusters. Therefore, the use of 1024 projects is preferred since this reduces execution time and produces no known negative side effects on the outcome of the model.

It is interesting to note that the optimum agent-to-project ratio is less than 1. This seems to support the hypothesis that FLOSS developers are a limited resource in the open source development environment. There are far more projects than there are skilled developers. The survival, and ultimately the success, of a project therefore depends on how well it can compete with other projects to attract the attention of developers from a rather limited pool.

7.3 Scenario Analysis

To gain a better understanding of the FLOSS development process, FLOSSSim is used to explore several scenarios. The scenarios were chosen based on their perceived ability to provide valuable knowledge about the FLOSS development process. For example, while there are many studies on FLOSS developers, almost no work has been done to understand the impact consumers have on open source software. The effect of consumers on the open source development process is considered in Section 7.3.1.

While FLOSS has grown in popularity over the last several years, it is likely the rapid growth is unsustainable. What will happen to the landscape of FLOSS projects if the rate of growth reduces? The results of reducing the rate of creating new projects is explored in Section 7.3.2.

Finally, core developers are frequently cited as a necessary component for FLOSS success. Unfortunately, core developers are a limited resource. Understanding when core developer contributions should occur in order to maximize the chances of project success is evaluated in Section 7.3.3.

7.3.1 Effects of Consumers

As previously mentioned, the effects of consumers on the FLOSS development process has largely not been studied. To further explore this topic, three scenarios are considered. In the first scenario, consumers are given the ability to use different selection criteria than producers when selecting projects. In the second scenario, consumers abandon the utility function altogether and instead choose projects randomly. In the third scenario, consumers are completely eliminated from the model. The results of each of these scenarios is then compared to the default model's results to reveal the impact consumers have on the FLOSS development process.

7.3.1.1 Separate Selection Criteria for Consumers and Producers

The base version of FLOSSSim assumes that producers and consumers use the same criteria when selecting projects. This may not, in fact, be the case. Passive users, for example, may be interested in software that is user-friendly, well-documented, and easy to install. If software requires a substantial investment in time, energy, etc., in order to use, it may simply be rejected by potential users. On the other hand, developers likely are interested in

a challenge. By definition, individuals interested in developing FLOSS seek projects that have tasks remaining, while passive users probably prefer projects that are fully functional and complete.

To allow consumers and producers the ability to select projects based on different criteria, the utility function must be modified. To allow for easier comparisons and to avoid introducing side effects, it is assumed that consumers evaluate projects based on the same five factors as producers, namely similarity of the project to the consumer, current popularity of the project with developers, cumulative size of the project, popularity of the project with other users, and maturity of the project. However, the importance assigned to each of these factors may differ between the two groups. Therefore, the utility function for the two groups remains as in (6.2) with the exception of the weights: there is now a set of weights w_{p1}, w_{p2}, w_{p3}, w_{p4}, and w_{p5} for producers and w_{c1}, w_{c2}, w_{c3}, w_{c4}, and w_{c5} for consumers.

FLOSSSim does not categorize agents as producers or consumers; rather, agents fall somewhere in the continuum between passive users and core developers based on their consumer and producer numbers. To avoid the problem of artificially assigning each agent to a category, and thus statically assigning a utility function for each agent to use, agents simply use the producer weights when producing and the consumer weights when consuming.

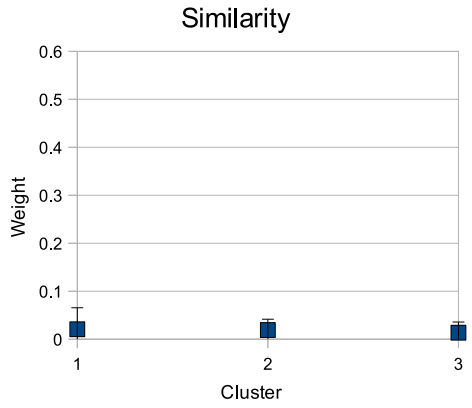
To see the effect of consumers using their own selection criteria, an evolutionary run is performed, allowing the producer and consumer utility weights to evolve separately. The producer weights may then be compared to the weights evolved in the base model to see if there is any effect from splitting producers and consumers. Likewise, the consumer weights can be compared to the producer weights to see if there are fundamental differences in how these two groups select projects.

TABLE 7.7
Producer utility weight cluster sizes.

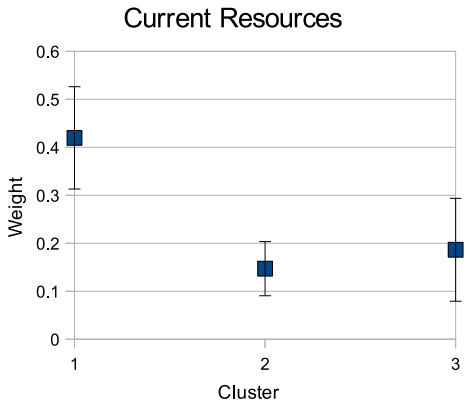
Cluster	Size	(Percent)
1	9	(22%)
2	17	(41%)
3	15	(37%)

The fitness values of the top 1% of parameter sets evolved is very similar to the base version of the model, with the minor differences attributable to the normal jitter present in the genetic algorithm and the stochasticity built into the model itself. Adding a separate set of utility weights for consumers has neither a positive nor negative effect on the ability of the model to match the empirical data.

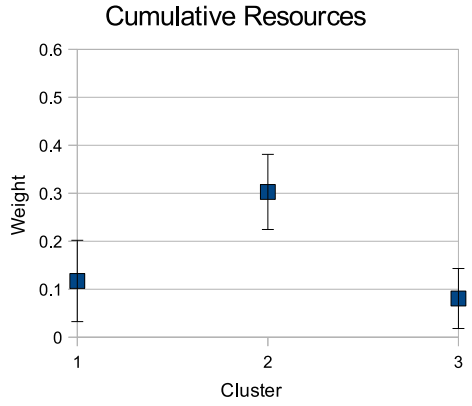
Cluster analysis on the producer weights shows the correct number of groups is three, allowing for direct comparison with the base version of the model. However, the producer weight clusters are less tight when separate consumer utility weights are included as compared to the combined weight clusters in the base run, indicating the producer weights do not cluster as well when consumers are afforded their own utility function. The increased variance may be explained by the additional five consumer weights w_{c1}–w_{c5}, which add degrees of freedom to the model, potentially resulting in more variability to the model's fitness score. The clustered producer weights are shown in Fig. 7.19 and the cluster sizes in Table 7.7. The weights for all three clusters are similar to the weights from the base model, as can be seen by comparing Fig. 7.19 to the base model's clusters shown in Fig. 7.8. As with the base model, once again the clusters are defined by the current resources, cumulative resources, and number of downloads, with each of these assuming a high value in a different cluster. The similarity and maturity weights remain relatively constant across all clusters, with similarity's importance being low and maturity's importance being high, sim-



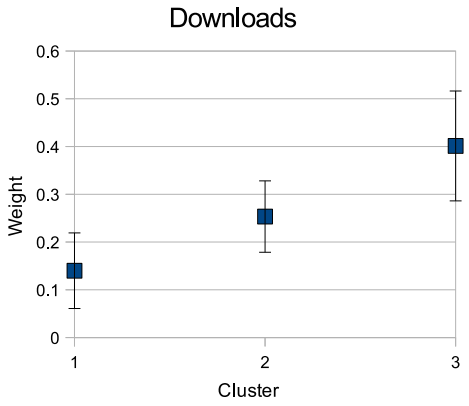
(a) Producer w_{p1} utility weight clusters.



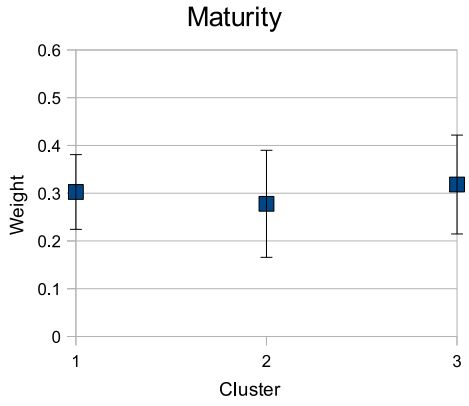
(b) Producer w_{p2} utility weight clusters.



(c) Producer w_{p3} utility weight clusters.



(d) Producer w_{p4} utility weight clusters.



(e) Producer w_{p5} utility weight clusters.

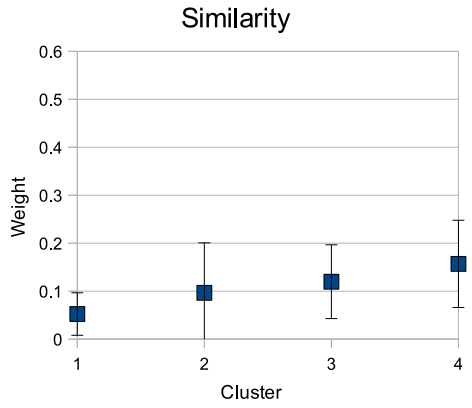
Fig. 7.19. Producer utility weight clusters when producers and consumers use separate utility weights.

TABLE 7.8
Consumer utility weight cluster sizes.

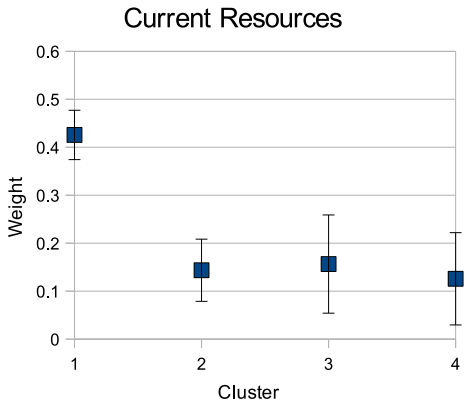
Cluster	Size	(Percent)
1	10	(24%)
2	10	(24%)
3	13	(32%)
4	8	(20%)

ilar to the results of the base model. Essentially, splitting consumers and producers results in minimum change to the producer utility weights. This indicates that consumers either use the same weights as producers when selecting projects or in general have a minimal effect on the FLOSS development process.

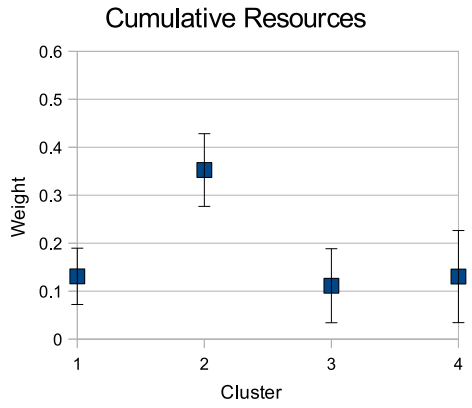
The evolved utility weights for consumers may be compared with the weights for producers to determine if there are inherent differences in the way consumers and producers select projects. The evolved utility weights for consumers cluster into four groups, as shown in Fig. 7.20, with cluster sizes shown in Table 7.8. The most noticeable difference when comparing the consumer weights in Fig. 7.20 to the producer weights in Fig. 7.19 is that the number of clusters differs. It appears that there are more profiles for consumers. In particular, the profiles for clusters 1, 2, and 3 are similar between consumers and producers in that both represent agents that value a project's current resources, cumulative resources, or downloads respectively when selecting projects. Consumers have a fourth profile, seen in cluster 4, that represents individuals most interested in the maturity of a project. Interestingly, similarity, which producers seem to almost completely ignore, receives a noticeably higher weight with consumers, indicating consumers are more concerned about selecting projects that match their interests. This makes sense. Whereas surveys have shown that developers have a wide range of reasons for being involved in FLOSS – to solve a problem,



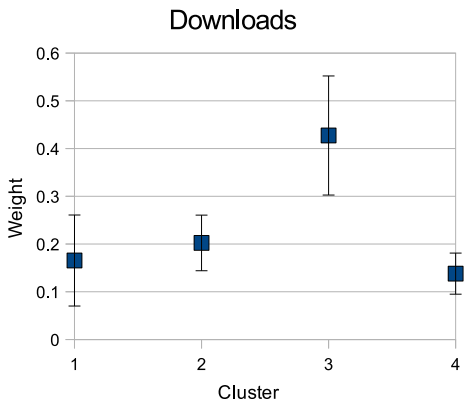
(a) Consumer w_{c1} utility weight clusters.



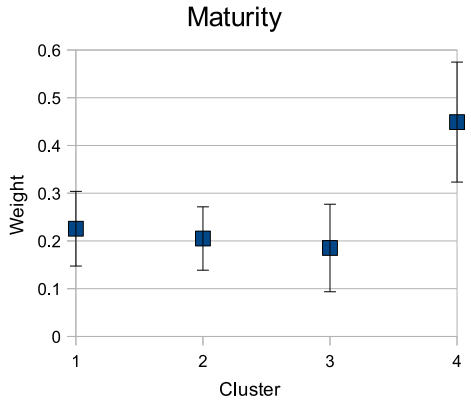
(b) Consumer w_{c2} utility weight clusters.



(c) Consumer w_{c3} utility weight clusters.



(d) Consumer w_{c4} utility weight clusters.



(e) Consumer w_{c5} utility weight clusters.

Fig. 7.20. Consumer utility weight clusters when producers and consumers use separate utility weights.

to maximize reputation, to learn/share skills, to improve job opportunities, etc. – consumers are typically simply looking for software solutions to their problems. A user's focus is therefore on finding software that is similar to his/her needs, hence the similarity weights being higher for consumers. Finally, it should be noted that in general the consumer weights do not cluster as well as the producer weights, as demonstrated by the on-average larger error bars. It is worth noting that everything else being equal, a larger number of clusters should result in tighter clusters, yet this is not the case when comparing the producer and consumer clusters. The lack of tight consumer clusters indicates that a larger range of values performs well for consumer utility weights. Compared to the tighter clusters of producer utility weights, this implies that consumers have a smaller effect than producers on the model's ability to match the empirical patterns.

In conclusion, the fitness values seem relatively unaffected by including separate consumer utility weights, meaning the model is able to equally reproduce the empirical patterns with or without this change. The fact that the consumer weights are not very stable and that the producer weights still settle into values similar to the default run indicates that consumers have a minimal effect on the model. Furthermore, the evolved weights for consumers and producers are largely similar, with three out of four profiles overlapping between the two sets, indicating that using a single set of weights may be a sufficient approximation. The return on investment for including separate utility weights for consumers is therefore not considered sufficient to warrent the added complexity to the model.

7.3.1.2 Random Project Selection by Consumers

To gain a better understanding of the impact of consumers, the model is modified so that consumers choose projects randomly, where each project in an agent's memory is chosen

with an equal probability. If this change has minor or no effect on the model's ability to match the empirical data, it indicates that consumers have a minimal impact on the model.

When parameter sets from the top 1% of the base model runs are rerun with a version of the model where consumers choose projects randomly, the fitness scores drop significantly. While the maturity fitness remains roughly the same, the ability of the model to match the developers per project and projects per developer distributions suffers. Consumers randomly selecting projects does not have a direct effect on developers, but if developers are considering the number of downloads as a reason to select a project, then this change indirectly affects them. Specifically, random selection evens out the number of downloads projects receive, resulting in a more uniform spread of download counts across all projects rather than the skewed distribution observed in FLOSS (as seen in Fig. 7.6b). Based on the utility weights evolved in the base model, developers do take the number of downloads into consideration when selecting projects, as seen in Fig. 7.8d. Therefore, changing the distribution of downloads will affect developers' choice of projects, which then impacts the model's ability to match the empirical data.

To determine if random selection of projects by consumers is more realistic than consumers using a utility function, an evolutionary run is performed using a version of the model where consumers choose projects randomly. Compared to the base model, where consumers use a utility function, the best evolved fitness values are not as high when consumers select projects randomly. Since the genetic algorithm fails to evolve parameter sets that perform as well in this scenario, consumers randomly selecting projects is rejected in favor of utility-based selection, indicating intelligent selection by consumers is likely what occurs in the real world since random selection results in a worse match with the empirical data. However, the fact that changing how consumers select projects has an effect on the

TABLE 7.9
Comparison of fitness values of the top 1% of parameter sets when there are no consumers versus the base version of the model.

	No Consumers	Base Version
Fitness Range	$[0.99931, 0.99943]$	$[0.99944, 0.99956]$
Average combined fitness	0.99935	0.99947
Average maturity fitness	0.99935	0.99939
Average developers per project fitness	0.99967	0.99991
Average projects per developer fitness	0.99902	0.99912

model's ability to match empirical patterns indicates that consumers have at least a minor effect on the model.

7.3.1.3 No Consumers

This scenario tests the influence of consumers on the model by eliminating agents from downloading projects. This is accomplished by simply setting all agents' consumer numbers to zero. The input parameter set is then re-evolved and compared to the default parameter set. If the model is able to produce similarly high fitness values without consumers, this indicates that consumers may not be important for matching the evolved patterns and therefore may, in general, be only minimally influential in the FLOSS development process.

The evolved fitness values for the top 1% of best performing parameter sets for the no consumers scenario and the base case are contained in Table 7.9. Without consumers, the evolutionary run fails to find parameter sets that results in fitness scores as high as for the default version of the model. In fact the worst fitness scores from the default model run are better than the best scores when there are no consumers, indicating the differences in fitness scores is likely a result of the change to the model rather than jitter due to the stochastic nature of the model runs and genetic algorithm. While all components of the combined fitness score drop when there are no consumers, the developers per project com-

TABLE 7.10
Comparison of the average evolved producer-related input parameters from the top 1% of parameter sets when there are no consumers versus the base version of the model.

	No Consumers	Base Version
Maximum number producing	12	12
Producer number mean	0.9647	0.9285
Producer number stdev	0.0870	0.0613

ponent is reduced the most. The developers per project is a highly skewed distribution, and including downloads in the model helps reproduce the skewed characteristic. In the default model, agents are able to take into account the number of downloads a project has received, resulting in developers flocking to highly-downloaded projects and thus yielding a skewed developers per project distribution. When there are no consumers, all projects have the same number of downloads (namely zero), meaning this is no longer a way to differentiate between projects, resulting in a less skewed developers per project distribution. Examining the parameters evolved for producers (consumer parameters are not relevant in this model run since consumers have been eliminated) shows that similar values have been evolved as in the base run, as seen in Table 7.10, demonstrating that eliminating consumers does not affect the frequency of developers contributing; rather, the main change in the model's ability to match the empirical data stems from how developers choose projects without the downloads information. Without this information, the model's ability to match the empirical patterns decreases.

7.3.1.4 Conclusion

The influence of consumers on the FLOSS development process has not been well studied. Using FLOSSSim to explore the impact of consumers, it is shown that users have at least a minor, indirect impact on the process. Namely, consumers are able to influence which

projects developers contribute to by affecting the download count of a project. In some ways this can be thought of as consumers voting for a project. Developers may only use the download count as a small part of deciding which projects to work on, and downloads may be outweighted by other factors, such as the current resources being contributed to a project. However, as long as w_4, the downloads weight in the utility function (6.2), has a non-zero value (it ranges from an average of 10% to 25% of the utility in the clusters for the base run; see Fig. 7.8d), consumers are able influence, albeit possibly very weakly, developers. Furthermore, FLOSSSim is able to most closely reproduce empirical data when consumers intelligently select projects, using either a combined or separate utility function when making decisions. When consumers choose projects randomly or are removed from the model entirely, the ability of the model to match the empirical data drops slightly, indicating users do have a minor effect on the model's ability to match the empirical data. However, the ability of the model to match empirical data almost as well when consumers are eliminated indicates that FLOSS should be considered a developer-driven process.

7.3.2 No New Projects

What happens if the rate of new projects being created is significantly reduced or reaches zero, and only the already-existing projects remain for developers to work on? This scenario is already occurring with Wikipedia, which essentially employs open source processes to write articles instead of software. In the case of popular languages, the majority of the mainstream articles have already been created and written, with largely specialty topics that require rare expert input or new topics, such as current events, new products, etc. being what remains to be created and written [208]. This has resulted in a tapering off of the number of articles being added to Wikipedia [208], [209], [210], and there has been a shift from

creating new articles to improving the quality and content of existing articles [208], [210]. Similarly, as there become good, high quality FLOSS solutions to common problems, the rate of new projects being created may be significantly reduced, with new projects largely falling into specialty categories. Likewise, the focus of open source developers may also shift, from creating new projects to collaborating on existing projects in order to improve and enhance them without having to reinvent the wheel. Indeed, as this occurs software may become less about creating new projects from scratch and more about assembling the existing, already functioning projects into larger and more complex software systems, all with reduced software team sizes [211].

If the FLOSS project creation rate significantly slows or stops completely, what will happen to the dynamics of the remaining projects? Will the cream of the crop – that is, the most attractive projects – at any given time simply float up to the top maturity stages as developers frequently contribute to these projects? Once complete, will the new cream (although this cream will be slightly less desirable than the previous projects in terms of attractiveness) be the next set of projects to be propelled to high maturity levels? Or will developers spread their distributions more evenly across the remaining projects when there are no longer good projects that stand out in the landscape of all projects? Once a particular project matures, will other similar projects simply be ignored since there already exists a good software solution to address this particular problem?

To consider the effects of a decreasing rate of growth, FLOSSSim is first run for 250 steps like in the default version of the model, with projects being added at a normal rate. FLOSSSim is then allowed to run for another 500 steps with two different treatments. In one case, projects continue to be added as normal in the simulation run, representing a base case. In the other case, no new projects are added for the remaining 500 steps. The

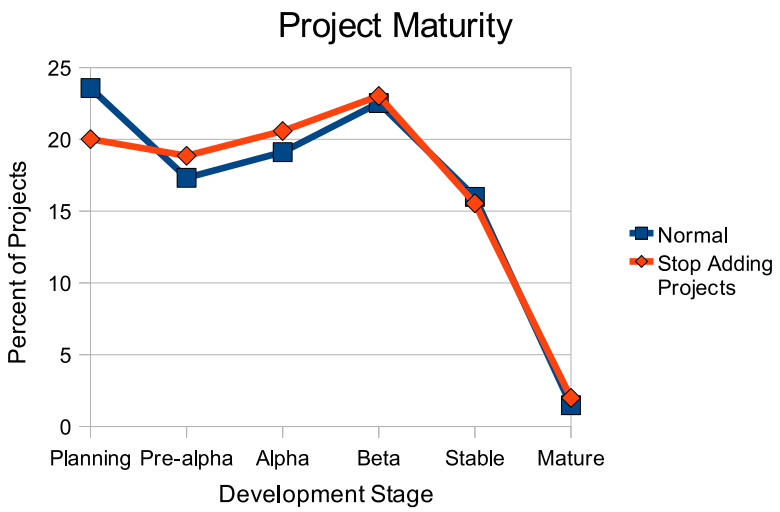


Fig. 7.21. Change in the percentage of FLOSS projects in development stages when no new projects are created for 500 time steps. When no new projects are added, projects in the early development stages progress to higher stages while there is minimal change for projects already in advanced stages. The data shown is based on a sample parameter set from the top 1% of evolved sets.

results of the two categories of runs are then compared to see how the project dynamics change in the extreme case when no new projects are created. Runs are repeated multiple times using parameter sets from the top 1% of the evolved parameter sets and the results are both averaged and inspected for general trends across multiple parameter sets.

The most noteable change can be seen in the maturity stage distribution. Projects in the planning, pre-alpha, and alpha stages receive more contributions when projects are no longer added, pushing the number of projects in the planning stage down and increasing the number of projects in the pre-alpha and alpha stages, as can be seen in Figure 7.21. The percentage of projects in the beta, production/stable, and mature stages is largely unaffected. The lack of change in the upper development stages between the two scenarios demonstrates a lock-in effect for successful projects. Through their contributions, developers influence successful projects to better serve their interests by changing the project's

needs vector. This serves as a reinforcing process that causes successful projects to remain successful. While the effect may be small due to the low value of the weight associated with similarity, it does increase the attractiveness of the project to the existing developers. In addition, by being popular with developers, other developers are more likely to be attracted due to the utility function's current resources weight, w_2. Likewise, as the amount of cumulative resources grows, the project becomes more desireable due to the w_3 utility weight. Combined, these properties ensure that developers continue to work on successful projects regardless of whether or not new projects are being added

The interest in lower development stage projects comes from the maturity score assigned to each maturity stage used in the utility function (see (6.4)); planning is by far the most important stage (i.e., it has the highest score by a significant margin), and pre-alpha is also important. In the scenario where projects are no longer being added, agents have less diversity in the projects to choose from; that is, the other factors influencing the utility function, such as current resources or number of downloads, may be insufficient to differentiate between projects. Thus the maturity becomes the discerning factor when selecting a project. Note, however, that due to the maturity scores assigned in (6.4), developers favor projects in the planning and pre-alpha stages. Once projects progress to the alpha stage, the maturity score is significantly reduced, and developers abandon projects in the alpha stage or higher in favor of other lower development stage projects, as demonstrated by the fact that the percentage of projects in the top three maturity stages remains largely unchanged.

In general, a change in maturity has a larger influence on the utility of a project than a change in the other components of the utility function. For example, when a project receives additional resources, increases the cumulative work, or is downloaded, in most cases this results in a relatively small change because each of these terms in the utility func-

tion is normalized with respect to the largest value in the entire simulation. For example, a project receiving a few downloads in a time step results in only a small effect when the project's total downloads is normalized to the maximum number of downloads any project has received[5]. On the other hand, because there are only six discrete development stages, moving between adjacent stages has a large effect on the utility as a whole, especially in the lower stages where the change in scores between adjacent stages is most significant. This means that in the situation where there are no new interesting projects being added and all the existing projects are no longer desireable (or at least there are no projects that stand out as significantly more desireable than others), the maturity term of the utility function will tend to control the overall perceived utility of projects, hence the noticeable change in the maturity distribution seen in Fig. 7.21.

Low development stage projects progressing while high stage projects remain largely stagnant highlights an important point, namely that a project must be desirable beyond simply being in a low maturity stage in order to progress long-term. A project must have another strong component in the utility function (i.e., a large number of current resources being contributed, a large amount of work accumulated, or a large number of downloads) in order to maintain receiving contributions. If a project is only attractive because it is in the early stages of development, the contributions from developers will not be sustainable, since the project will become less and less attractive as the development stage increases. It becomes clear then that projects that are in high development stages must

---

[5]This is because for project i at time t,

$downloads_{i,t} - downloads_{i,t-1} << max(downloads_{1,t}, downloads_{2,t}, \ldots, downloads_{numOfProjs,t})$

in most instances.

overcome the lower utility they receive from the maturity component of the utility function. If these projects are able to offset the low maturity score because of one or a combination of high current resources, cumulative resources, or downloads, then they will likely reach critical mass and progress to upper development stages or even project completion. On the other hand, low maturity stage projects may attract developers due to the high scores assigned to the early development stages, but as they mature, this drops off and the projects are abandoned unless they have some other component of the utility that is stellar and can overwhelm the drop off in utility from maturity. For example, a young project might accumulate developers while it is in the lower stages, causing the current number of resources term of the utility score to grow fast enough to offset the shrinking maturity term. More often than not, this is not the case. This behavior is in agreement with [129], which argues that projects need core code already written when released as open source in order to attract contributors. FLOSSSim mimics this because some of the projects that start in advanced stages excel. Meanwhile, projects in early stages may gain some contributions, but these will likely drop off as the project matures, resulting in a stagnant and incomplete projects.

The percentage of projects that are being worked on (i.e., projects that have at least one developer associated with them) also changes between runs, with an average of 12% of projects being worked on when new projects are being added versus 23% when no new projects are being created. At the same time, there is a small shift in the number of developers per project distribution. Namely, when projects are no longer added, the percentage of projects with a single developer decreases. Note that this distribution is normalized to include only projects with one or more developers, meaning if the percentage of projects with a single developer decreases, the percentages of projects with more than one developer must increase. Indeed, the bulk of the change is seen in projects with two or a few

TABLE 7.11
Change in the average percentage of projects with N developers when no new projects are created for 500 time steps. It is slightly less common to have single developer projects in favor of projects with just a few developers when new projects are no longer created. The data are averages from runs with the top performing parameter sets.

Developers per Project	Normal	Stop Adding Projects
1	76%	68%
2	12%	16%
3	4%	5%
4	2%	3%
5	1%	2%
6	1%	2%
≥ 7	4%	4%

developers, where the percentage of projects in these categories has increased slightly, as can be seen in Table 7.11. Considering that the number of agents remains constant, at first it seems counterintuitive to observe an increase in the number of projects being worked on and at the same time have the number of single developer projects reduced and replaced with two or more developers projects. However, this can be explained as follows: when projects are being added, there are occasionally highly desireable projects created. Some developers will flock to these projects while the remaining developers, who might not be aware of or view these projects as desirable, will tend to spread themselves out over many different projects. When new projects are no longer being added, the projects that some developers found desirable are steadily worked on and quickly become either undesirable (e.g., the projects gradually become less desirable as they move up in development stage as a result of (6.4)) or ineligible for additional contributions (i.e., the projects are completed). The developers who worked on these projects must now find other projects to contribute to. Unfortunately, no new desirable projects are being added to the simulation, so these developers must join the other developers in choosing from among the remaining,

less-interesting projects. This results in a larger number of projects having developers associated with them because there are no longer clusters of developers working on certain very attractive projects; rather, the developers' contributions are spread across more projects. However, as shown by the change in the maturity distribution, because of the influence of the maturity weight in the utility function, the bulk of developers are really only selecting from projects in the lower development stages. When there were desirable projects, this didn't necessarily happen, since a project with, for example, a high cumulative resources score might have a higher utility value than a project in the early development stages with a small cumulative resources score. The tendency to select from only a subset of projects when there are no particularly desirable projects available results in more projects within this set having a few rather than just one developer. Essentially this is an extension of the pigeon hole principle, which shows that as the number of pigeon holes is reduced (in this case the number of projects developers are likely to choose from), the chances of multiple pigeons being placed in the same pigeon hole increases (that is developers choosing the same project).

Other distributions, including the projects per developer and downloads, are largely unaffected when new project creation ceases during a model run.

In conclusion, FLOSSSim provides predictions of what will occur in the FLOSS landscape if the number of projects being created significantly decreases. Based on the change in the maturity distribution, it appears that many early stage projects will at least get started, with developers working on these projects enough to propel them to slightly higher development stages. However, these projects will fail to maintain the interest of developers, as demonstrated by the fact that the percentage of high development stage projects does not grow. Meanwhile, it is predicted that the really good projects, those that are very attractive

to at least a subset of reliable developers, will progress to the advanced stages regardless of whether or not new competing projects are being added, as shown by the similar percentage of projects in the upper development stages in both runs. This is important because it shows that projects that are uninteresting when there are new projects being added remain uninteresting when no new projects are being added as well. Furthermore, while some of these uninteresting project will progress when there aren't new projects being created, and at the same time the average number of developers per project will also increase, this won't necessarily result in good, usable, mature projects materializing. This behavior mimics the real-world phenomena where a single project in a particular category becomes the "winner," or the default software to use to solve a particular problem. There will be other projects in the same category that at least get started on development, but these projects most likely will not mature into good, useable software, let alone software that can compete with a similar, already-established, well-functioning project.

7.3.3 Effects of Core Developers

Core developers may play a key role in the probability of FLOSS projects becoming successful. Indeed, a project with no core developers likely will never achieve success, even if there are many contributions from peripheral developers. This is partially because core developers act as the glue that holds a project together. They are responsible for joining together the contributions to form a coherent, quality project. Even Linus Torvalds, one of the best known core developers in the open source community, admits that the majority of his time is spent piecing together the components and performing quality control and guidance to Linux [34].

As projects mature, the complexity of the projects also tends to increase. The core developers, who have worked on a project long-term, accumulate knowledge and may be the only ones who sufficiently understand the software's design in order to provide guidance to move the project forward [212]. If a key contributor leaves a project, knowledge is lost and the project may eventually fail as a result [187]. Based on this observation, are there both good and bad times for a core developer to leave a project, some of which will allow a project to survive and others which almost always result in failure? If a core developer can contribute to a project for only a finite amount of time, when is the most advantageous time in a project's lifecycle to have the core developer involved? For example, if a core developer is involved in the early stages of project development, will this result in sufficient interest from other contributors, possibly even causing other core developers to join the effort, such that the project is self-sustaining even after the core developer leaves? How robust is the FLOSS development process to core developers joining and leaving projects?

To explore the effect of core developers contributing to projects at different times during development, FLOSSSim is modified to add and remove core developers to projects at specific stages of development. Core developers are modeled simplistically as agents that consistently contribute substantial resources to a project; knowledge and coordination skills are not modeled. Three treatments of core developers are considered: core developers involved in projects during the early stages of development (i.e., planning and pre-alpha), the mid-stages (i.e., alpha and beta), and the late stages (i.e., production/stable and mature). The success of projects subjected to the three treatments is then analyzed to help understand 1) at what stages it is most important for a project to have a core developer, and 2) when core developers can be removed from a project while still leaving the project with a good chance of survival.

The setup for testing is as follows: In a simulation run, a subset of projects are randomly selected to be the recipients of core developers. The run is then executed four times with the same random seed. One run acts as the control case, where the selected projects do not receive contributions explicitly from assigned core developers. The remaining three runs add a single core developer to each project when the project is in the planning and pre-alpha, alpha and beta, or production/stable and mature development stages. While a project is in the selected stages, the assigned core developer consistently adds a 40 hour week worth of work to the project (i.e., 1.0 resources are contributed) at each time step. Outside of the selected development stages, the core developer does not work on the project. While contributing, a core developer's needs vector is made to perfectly match the needs vector of the project the developer is working on, representing the very similar interests one would expect between a project and a highly dedicated developer working on that project. For each experimental run, FLOSSSim is configured using parameters from the top performing 1% of evolved parameter sets. Multiple runs are performed to accommodate for the stochasticity in the model and the results averaged. The number of projects selected to receive core developers during each run is set at eight in order to achieve a balance between the number of runs necessary to collect sufficient data and so as not to saturate the landscape of FLOSS projects (in this case, fewer than 1% of projects have core developers artificially assigned to them; if too many projects are artificially assigned core developers, this will alter the dynamics of the model). After 250 time steps each of the eight projects is checked to determine if it is successful according to the six success metrics described in Section 7.1.4.1.

The average percentage of successful projects for each of the core developer contribution periods is shown in Fig. 7.22. According to four of the success metrics, namely *ma-*

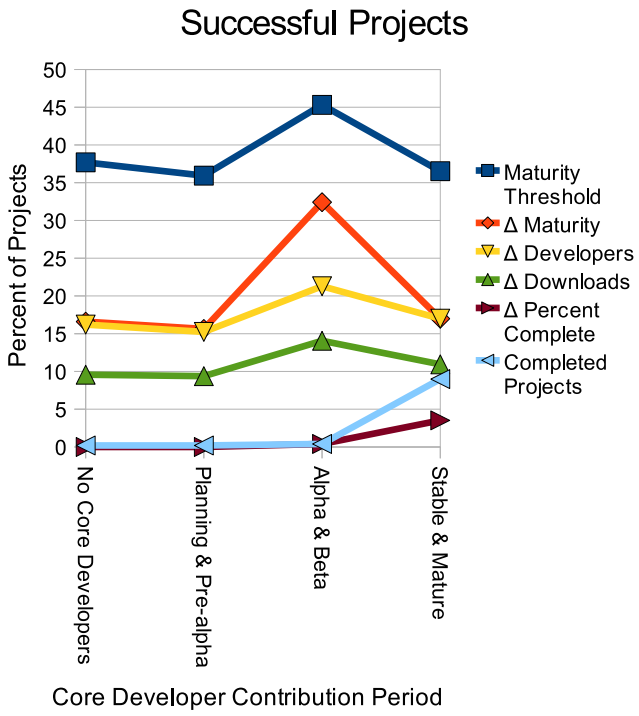


Fig. 7.22. Effects of core developer contribution period on project success. Projects are most likely to succeed when a core developer is involved during the alpha and beta stages of development.

turity threshold, Δ *maturity*, Δ *developers*, and Δ *downloads*, projects have the best chance of being successful if core developers are involved during the alpha and beta stages of development. As a reminder, the success metrics are collected at the termination of a simulation run, after 250 time steps. This means that regardless of when during the simulation run the project was in the mid-development stages, at the end the project was considered successful according to $2/3$ of the success metrics. In some cases this is expected. For example, the *maturity threshold* metric categorizes all projects in the beta development stage or later as successful. When the core developers are involved in the alpha and beta stages, any project that advances to the alpha stage will continue to advance and thus likely be categorized as successful by the end of the run. On the other hand, the remaining three success metrics that

are highest when contributions are made in the alpha and beta stages show that a number of projects are still advancing in maturity, gaining new developers, and being downloaded at the end of the simulation, even though they may have advanced past the stages where core developers contribute to the project. This indicates projects may be sustainable after core developers leave, so long as those core contributions were made in the mid-stages of development.

It is likely no conincidence that the most important stages to have a core developer associated with a project are also the stages where the bulk of the commits occur (see Fig. 6.4 on page 174). To determine if there is a difference between the core developers contributing in the alpha versus beta development stage, the analysis is repeated two more times, once with core developers contributing only in the alpha stage and once only in the beta stage. For the four metrics that peaked when core developers were involved in the middle development stages, the results were mixed. The *maturity threshold* metric had the highest percentage of successful projects when core developers were involved in the alpha stage while the Δ *maturity* and Δ *downloads* metrics were highest when core developers were involved in the beta stage. The Δ *developers* success metric showed a similar increase when core developers were involved in either stage.

One might expect that if more projects are successful when core developers are involved during the mid-development stages then there would also be an increase in the number of successful projects when developers are involved in the late stages of development. This is not the case for the four metrics that peak in the alpha and beta stages. Instead, the percentage of projects that are considered successful when core developers are added in the production/stable and mature stages is essentially the same as when no core developers are explicitly assigned to projects. This is because these projects never make it to the upper

development stages to collect the resources of the core developers. Instead, the projects stall in lower stages. Thus, core developer contribution during the alpha and beta stages can be seen as helping a project at a critical time, propelling the project through a lull that bridges the gap between when a project is new and when it is mature.

The remaining two success metrics, Δ *percent complete* and *completed projects*, perform best when core developers are involved in the final stages of development. It is not surprising that the number of completed projects increases when core developers are involved at the end of the lifecycle. Normally, very few projects make it to the top stages and even fewer make it all the way to completion. Having a developer consistently contribute to projects in the upper stages will thus increase the number of projects in the top stages that manage to progress to completion. Furthermore, the percentage of commits that occur in the top stages is much smaller than mid-stages, resulting in consistent contributions from a core developer in the final stages rapidly moving a project through the stages as compared to the same size contribution in a mid-development stage. Thus it is even more likely that a project that reaches the upper-development stages will be complete by the termination of the simulation run if core developers join in the production/stable development stage.

Interestingly enough, having core developers involved at early stages of development does not seem to have a positive effect on the success of a project, with the percentage of successful projects being almost the same as when no core developers are assigned. As previously mentioned in Section 7.3.2, projects must overcome a number of obstacles to progress beyond the early stages of development, especially since the attractiveness of a project according to its development stage decreases as the project matures. It appears that even adding core developers at the early stages does not have a significant enough impact to offset the dropoff of appeal as projects progress. In fact, removing core developers at

the end of the pre-alpha stage may be the worst possible scenario, because a project loses appeal by moving into the alpha stage and the utility is further reduced because the current contributions are reduced. Indeed, several of the success metrics show a small decrease in the percentage of successful projects when core developers are involved in the early stages of development as compared to the control run, although the differences are small enough that they are likely a result of noise.

Note that this experiment was performed by adding only a single core developer per selected project. In reality there may be multiple core developers involved in a single project, and in some cases these developers may join or leave a project at relatively similar times. It is expected that the results will simply be more pronounced if groups of core developers join or leave a project concurrently.

In conclusion, the bulk of success metrics indicate core developers are most important during the alpha and beta stages of development. This makes sense, since not only does the bulk of work on a project occur during these phases, but it is also the core developer's responsibility to take not only their own work but the work of others and assemble the chaos into functioning software. Paired with evidence that projects should have at least a kernel of working code when they are released to the open source community, this provides a major advantage to projects that are first developed to a functional level cathedral-style and then released into the open source domain while keeping the original developer(s) involved in the project, at least temporarily, during the transitional period. These original developers act as core developers, at least until replacements can be found, and help the project navigate a critical part of the development lifecycle where many projects falter.

The analysis of the model also shows that the FLOSS development process is not very robust to adding and removing core developers. There are times during the develop-

ment lifecycle that it is more important to have core developers, namely during the alpha and beta stages of development. However, core developers that are involved during other stages of FLOSS development seem to neither influence the probability of success or failure of a project. This highlights the importance of core developers, which according to the findings presented here, are able to influence the success of a project. Developers willing to act as core developers are a limited resource in the FLOSS domain, and the shortage of core developers compared to number of projects in existence may help explain why so many projects never get off the ground.

7.4 Discussion and Future Work

An ongoing challenge with developing FLOSSSim has been finding methods to map real world data into the model. Efforts to find additional methods to accomplish this should continue in order to increase the realism and usefulness of the model. Building a model that could be seeded from real world data was an original design goal of this research. Ideally, this would mean a snapshot of the FLOSS landscape on, say, SourceForge, could be taken and fed into the model. FLOSSSim would then be capable of mapping the relevant project details from the snapshot into the simulated environment. By doing this, it would be possible to explore many additional interesting and important scenarios. For example, it would be possible to create a customized project, drop it into the simulated landscape, and monitor its progress. The simulation could be re-run multiple times, tweaking the project (or other projects) and looking at the long term effects of the changes. This would be immensely helpful in answering important questions about the FLOSS development process that are part of the motivation of this research, such as: Which projects will still be active in N years? Which projects will thrive and which will fail? What can be done to increase the probability that a project will be successful? Are projects the masters of their own destiny,

or does the surrounding FLOSS landscape have a significant impact in determining what happens with a project?

The ability to better map real world data into the simulated environment, especially at an individual project level, would take FLOSSSim to a level where individual projects can be studied. With this enhancement, it would be possible to study a project in a very specific scenario versus the more generic analysis that can be performed with the current version of the model. This could result in powerful prediction functionality. The bulk of the analysis conducted so far has focused on aggregate analysis, looking at trends across many projects, but not studying single projects in the landscape, let alone projects in very specific scenarios. FLOSSSim was designed and implemented this way partly because of the data available, which is mostly aggregate, and partly because a high-level model is the first step in creating a more detailed model.

How to move from a "coarse" to a "fine" model remains a long-term and open problem. Where possible, the model already uses empirical data, but even when data is available, sometimes it cannot be easily incorporated into the model. Essentially, one of the major obstacles is how to map real world data into the model's representation of that data (or, as an alternative, to change the model's internal representation of data so that it more easily maps to real word data). In some cases there is a natural mapping, but frequently model data is an abstraction and/or simplification of the real world phenomena, meaning there is not a trivial function to map between the real and simulated data. For example, how can the topic of a project posted on SourceForge be mapped into the abstract concept of a project's needs vector? One possible solution would be to use the topic categories and

subcategories that a project is listed in on SourceForge[6] and map these to the needs vector. This could be further extended to include other properties of a project as well that might be of interest to an agent selecting a project, such as platform, programming language, license, etc. One problem is that this may work for SourceForge, but combining data from other forges, which may have different categories or no categories at all, will still be a problem. In addition, there remains the constant concern about dirty data. For example, SourceForge permits projects to be listed in multiple topic categories, but lazy project developers may only bother listing their project in a single main category, even when other categories may also apply[7].

Another major challenge with enhancing the model's ability to be seeded with real world data is a lack of availability of the necessary data. For example, the needs and interests of consumers of FLOSS have not been measured well at time of writing. Finding a method that would allow for accurate measurement of this data is another open problem.

Regardless of the complications, much of the future work that can be done with FLOSSSim depends on providing a better mechanism to map real world data into the model. Because of the difficulty and size of this problem, this should be treated as a long-term goal that likely will be solved in many small, incremental steps. Even minor improvements

---

[6]See http://sourceforge.net/softwaremap/?&fq[] for a list of SourceForge trove categories.

[7]This may not be a problem for people looking for projects in the real world, who are likely to use keywords to find a project that meets their needs as an alternative or in addition to to browsing topic categories. Thus keywords found in project descriptions might offer another option for mapping the real world data to FLOSSSim, although this is not without complications as well, e.g., some projects may provide good descriptions on their project summary pages while others may place this information in README files or other documentation that isn't necessarily indexed by the site.

at a non-individual project level (e.g., adding more aggregate data) will help to improve FLOSSSim's usefulness, accuracy, and validity.

Dirty data has been major problem throughout the development of the model. Due to time restrictions and the extensive possibilities, an incomplete sensitivity analysis has been performed. Additional sensitivity analysis should be conducted, especially for data that is central to the model, to discover how robust the model is to minor data perturbations. For example, the maturity stage thresholds were calculated from data that, once filtered, resulted in a small and incomplete data set (i.e., very few project and no projects having progressed through all six stages). A sensitivity analysis should be performed to establish how critical these values are to the performance of the model. If the model performs well over a wide range of these values, the importance of obtaining exact values for the thresholds from the dirty data is reduced and confidence about the performance of the model with the current potentially imperfect data is increased.

An interesting enhancement would be to expand the model to use data from forges other than SourceForge. There are a number of complications with doing so, most noteably the heterogeneity of the data. Although SourceForge is the main source of data for FLOSSSim, in part because it is the largest and most popular forge and because using a single data source solves problems associated with heterogeneous data, there are some concerns whether or not the projects on SourceForge accurately represent the FLOSS domain. Part of the concern comes from certain categories of projects tending to be hosted elsewhere. For example, many successful projects host their project homepages and development tools on their own servers. Some of these projects may still maintain placeholders accounts on SourceForge, but more than likely the SourceForge data associated with these projects will be inaccurate due to the real data being hosted with off-site tools on an off-site host. Other

classes of projects might also be underrepresented on SourceForge. For example, the Free Software Foundation maintains the Savannah forge for official GNU software[8] and for other projects that meet the free software definition[9]. As mentioned in Chapter 1, free software is a slightly different concept than open source, although most of the licenses used by free software meet the definition of open source as well. This means that no official GNU software is hosted on SourceForge. Because GNU is a major player in the FLOSS domain, developing many high quality projects, and because the free software movement is slightly different than the open source domain, using data only from SourceForge will result in missing potentially important data.

Moving away from using data only from SourceForge and designing a better method of mapping individual project data into the model might go hand-in-hand. For example, much project-specific information can be extracted by analyzing SCM logs. Virtually all open source projects use SCM software, regardless of where they are hosted, and most of this software is configured to allow anonymous read access. Thus, it becomes a matter of determining the URL to interact with each project's SCM software. Fortunately, many forges use a set of naming conventions for creating a project's SCM URL so that with the right information, such as the project's name, the SCM URL can be determined. This is the approach the FLOSSMetrics project uses to collect data and has the major advantage of breaking away from a single forge while maintaining close to homogeneous data for

---

[8]http://savannah.gnu.org

[9]http://savannah.nongnu.org

projects[10]. Of course forges also track information about a project that is not contained in the SCM logs (e.g., project category), so becoming forge-independent may not be a possibility.

With enhancements allowing improved mapping of the landscape of projects into the model, it would also be possible to explore if there are major differences among the forges. The model could initially be calibrated using, say, SourceForge data, and then its performance measured when using data from other forges. If differences were detected across forges, understanding the differences could significantly contribute to the knowledge of what conditions foster successful FLOSS development.

Another possible enhancement to the model would be adding the concept of "capabilities" to agents. Essentially, the skills of developers are not all equal, nor are the competencies required to work on a project the same for all projects. Spending one hour on one project results in more progress than spending one hour on a second project if the developer is skilled in the tasks that need to be completed by the first project but inexperienced in those needed by the second. Indeed, a developer not possessing certain skills may be unable to make any progress on a project if there is only specialty work remaining. An example would be a project that requires the developer to be a subject matter expert for the particular domain. Developers who are not familiar with the domain may not be able to provide meaningful contributions. As implemented now, FLOSSSim uses resource numbers as an abstract representation of work. While agents are endowed with different

---

[10]SCM data is not quite homogeneous because different SCM software stores different data, or possibly the same data in a different format. However, because the purpose of all SCM software is the same, it is expected that there is more overlap than unique data stored by the different systems, meaning much of the data is universal across all systems.

resource numbers, there is no concept of efficiency in the work performed based on the tasks matching the skills of the developer. Instead, any agent contributing R resources to a project is the same as any other agent also contributing R resources to the same project. In both cases, the project progresses the same amount.

One possibility for adding capabilities to FLOSSSim involves modifying the needs vectors. The needs vector could be a bitstring representing all possible skills in the universe. Each bit would correspond to a single skill, with a one representing possession of the skill and a zero a lack of the skill. Agents would then have needs vectors that contained the skills they possessed. Likewise, projects would have needs vectors that contained the skills necessary to complete project tasks. Essentially, a project's needs vector would represent the desired functionality of the project and an agent's needs vector would represent what types of functionality an agent is capable of implementing. Agents and projects would need to have overlapping bits set to one in their needs vectors in order for an agent to be able to make a contribution to the project. Bits in a project's needs vector could be cleared when tasks involving the respective skill were completed and developers possessing the skill were no longer needed. In this sense a project's needs vector would represent the amount of work remaining on the project. When the needs vector only contained zeros, there would be no more tasks to complete and the project would be considered finished.

In the spirit of keeping the model as simple as possible, capabilities were not implemented in this version of the model, nor is there evidence that adding capabilities to the model will increase the model's performance. However, a number of peers who have reviewed the model have suggested capabilities be added to increase the realism.

Quality may be an important factor in motivating individuals to choose projects that is not currently incorporated into FLOSSSim. Arguably quality is of more importance

to consumers than producers because users are trapped with using the existing software as is, whereas developers have the ability to change the software and thus work around or improve the quality if it does not currently meet their standards. Consumers, on the other hand, are looking for solutions to their problems and a low quality solution may not sufficiently motivate a user to try the software. Similarly, trying a project and finding it is low quality may cause a user to steer clear of that project in the future. Indeed, [125] finds a positive correlation between software quality and FLOSS usage; in particular, increased quality leads to more FLOSS use and higher user satisfaction [125]. Higher user satisfaction also in turn increases FLOSS usage [125]. Even if quality is only important to users, the non-zero weight w_4 evolved for downloads in the FLOSSSim utility function indicates this may still affect the FLOSS development process. The concept of quality has already been incorporated into one other existing FLOSS model [37].

In reality, most FLOSS projects are never complete. The majority of projects that manage to release a stable version of software will continue to add new features and make other improvements. In many cases, as a project moves from a stable release to the next version, the software will regress in development stage, most likely moving from a production/stable or mature stage back to a beta or earlier stage. FLOSSSim does not include these dynamics. Instead, when a project is created one of its properties is resources for completion, a static number indicating the amount of work necessary to complete the project. When the cumulative resources equals the resources for completion, the project is considered finished. Assigning a resources for completion value to each project is necessary in the model in order to calculate the maturity stage based on the percent of the project that is complete. However, FLOSSSim could be modified so that projects occasionally increase their resources for completion number, representing developers adding new requirements

to the project, expanding the scope of the project, etc. A large increase would also result in a downgrade to the project's maturity, since the percentage of the project that is complete would be reduced. Allowing projects to increase in size and regress in maturity stage might change the dynamics of the model. For example, successful projects would attract the contributions of developers for longer since these projects wouldn't simply rapidly terminate but instead might proceed through several iterations of increasing in the resources necessary for completion.

The majority of FLOSS projects are not standalone software. Most projects have dependencies, some quite an extensive list, which are frequently other FLOSS projects. Some projects, such as software libraries, are never intended to be standalone but are always used by other software. Thus, there is an inherent link between many projects, meaning the welfare of one project may actually affect the prospects of the many other projects that depend on it. A possible enhancement to FLOSSSim would be to include links between projects that depend on one another. This would allow for studying network effects as they pertain to FLOSS project success. For example, how does the failure of a project affect all the projects that include it as a dependency? Is there a reverse ripple effect, where a popular project results in its dependencies also receiving more contributions, perhaps because developers from the popular project contribute to the dependencies to add functionality necessary for the popular project, or perhaps because being listed as a dependency serves as effective advertising for these projects? Thus by including dependencies, FLOSSSim could be used to explore network dynamics in the FLOSS domain and answer these types of questions. It

should be noted, however, that sites like SourceForge do not track project dependencies as part of a project's metadata and therefore collecting this information may be a challenge[11].

Once developers join a project, it is likely that they will continue to work on the same project in the future. This is especially evident in the case of core developers, who typically work on a project for an extended period of time. Currently, FLOSSSim attempts to reproduce this characteristic by giving a boost (taking the square root) of the utility function for projects worked on in the previous time step. In effect, this increases the probability of an agent selecting the same projects to work on in the subsequent time step. Improvements to the model might include adding a switching cost term to the utility function, as was done in FLOSSSimple, representing the extra effort required to become familiar with another project. An alternative solution can be seen in [58], which address this issue in their FLOSS model by using probabilities based on data from SourceForge to determine when developers continue working on or leave a project they are currently involved with.

The model's needs vectors serve as an abstraction for representing the interests and corresponding functionalities of the agents and projects respectively. Therefore, the needs vector is at the crux of handling the matching of developers' interests with appropriate projects. Because the actual distribution of people's interests is unknown, a simplistic approach is taken and the needs vector values in FLOSSSim are assigned via a uniform dis-

---

[11] One possibility that warrants investigation for efficiently collecting dependency requirements would be to look into "packages" and "package managers". Packages are used to make distributing software easier. Included in a package is a list of dependencies, and part of the package manager's job is to make sure these dependencies are also installed and, if not, possibly to fetch and install them as well. It is not uncommon for FLOSS projects to release versions of their software as packages for some of the more popular Linux distributions, although certainly not all FLOSS is distributed as packages. Thus mining packages may be a method for extracting dependency information for at least some projects.

tribution. However, exploration of the effects of other distributions may be interesting. For example, if a normal distribution is used, projects with vector components near the mean may have an advantage because there will be many agents with interests similar to the projects. Projects with vector components several standard deviations from the mean will have a much smaller set of agents whose interests are similar to these projects. A drawback of a normal distribution is that it makes most projects similar; in reality, projects are spread over a wide spectrum (e.g., from operating systems and drivers to business applications and games). The challenge centers around generating needs vectors that are distributed similarly to real people's needs. Unfortunately, the interests of those involved in FLOSS, especially consumers, is not known. It might be possible to use the number of already-existing projects in predefined categories as a proxy for people's interests and from this gain insight on how needs vectors can be generated to match reality. Indeed, topic categories on SourceForge such as Internet and Software Development have project counts two orders of magnitude greater than some of the other categories, such as Printing and Terminals [213]. The imbalance may indicate that those participating in the FLOSS domain are more interested in certain types of projects, such as web browsers, than in other types of projects, such as printer drivers. However, the skewed data may also be a result of uneven partitioning of the topics at the top level. Thus, determining the correct distribution for the needs vector remains an open, but important, problem since needs vectors are integral to the model. Changing the distribution used to generate the needs vectors likely would affect w_1, the evolved similarity weight in the utility function. It would also likely affect the analysis of target audience size versus success.

While projects' needs vectors evolve based on the contributions received from agents, agents' needs vectors remain static throughout a simulation run. In reality, agents'

interests may change over time and may even be affected by the projects they work on (i.e., an agent may learn to like or dislike a topic based on their experiences with a project). The ability of agents' needs vectors to evolve based on the projects they are involved in might increase the realism of the model.

In the real world, projects may be perturbed by exogenous events. For example, a feature added in one project may influence other projects. This phenomenon could be incorporated into the model by occasionally mutating a project's needs vector randomly.

Agents who work together may influence each others' choices in joining or leaving projects. Currently, when an agents' memory is updated, a randomly selected project is added or removed, but it may be more realistic if knowledge of projects is influenced by an agent's peers. For example, an agent contributing to project P would be more likely to discover (i.e. add to its memory) other projects that co-developers of project P are working on. Likewise, an agent developing for multiple projects may mention abandoning a project, influencing developers that agent interacts with while working on other projects. Occasional random selection of projects would still be required, as agents may still independently discover or reject projects. Implementing these changes may increase the membership herding dynamics occurring in the model.

A static number of agents is created at the beginning of a model run. Adding and removing agents during the execution of the model would increase the realism of the model. However, one caveat with this enhancement is that the rate at which to add and remove agents is unclear and cannot be easily determined from empirical data. For example, the number of developers on SourceForge appears to increase over time, but part of this may be because developers only register, but never unregister, with the site. There is no data available on consumers joining and leaving the open source community.

One final complication with the model is its internal representations versus reality. For example, a suggested strategy for success in open source projects is to release early and release often [13]. Using this method to determine successful projects within the model is problematic because the model includes no concept of releasing versions of software. Augmenting the model to include a reasonable representation of software releases is non-trivial, if possible at all. Likewise, it is difficult to compare findings of other work on conditions leading to success that map into this model. For example, [101] consider licensing impacts while [81] considers version control systems, mailing lists, documentation, portability, and systematic testing policy differences between successful and unsuccessful projects. Unfortunately, none of these aspects easily map into the model for comparison or validation purposes.

7.5 CONCLUSION

A better understanding of conditions that contribute to the success of FLOSS projects is a valuable contribution to the future of software engineering. An agent-based model was created to explore these conditions. The model is formulated from empirical studies and calibrated using multiple sources of FLOSS data. The calibrated version reproduces distributions that closely match the three emergent properties examined. It is shown that these distributions are non-trivial to match. In addition, the model is able to predict distributions it was not calibrated for, adding confidence that the model is valid. From the calibrated data, it is concluded that the current resources going towards a project, the resources a project has already accumulated, the number of downloads a project has received, and the maturity of a project are all important factors when selecting projects. Surprisingly, the similarity between an agent and a project is not important in matching the empirical data. Cluster analysis of the best performing parameter sets shows that there may be several different

classes of agents, each with a different primary interest driving their selection of projects. The notion that FLOSS is primarily a developer-driven process is supported, while users are a distant second in influencing the process. Finally, different definitions of success are explored. It is concluded that different formulations of success result in different sets of projects being considered successful, with minimal overlap between pairs of sets. The size of the target audience does not play a major role in influencing the success of a project.

The model is also used to evaluate several scenarios. Through this analysis the influence of consumers, a group which has not been well-studied, is further explored. It is reaffirmed that consumers play only a minor role in influencing the FLOSS development process, and there is insufficient evidence to show that consumers use significantly different selection criteria than producers. In addition, it is shown that while maturity stage has a large influence on which projects receive contributions, projects must be desirable beyond their development stage if they are going to progress to completion. Projects that only score high in one of the factors in the utility function will likely only perform well for a short period of time and then burn out. Furthermore, some projects are inherently desireable. Eliminating these projects does not make the previously undesirable projects any more desirable, as illustrated by the fact that these projects do not progress to upper development stages even when only uninteresting projects remain. Finally, it is found that core developer participation is most important in regards to influencing project success if it occurs in the mid-stages of development; projects that attract core developers during the mid-stages may become self-sustaining and survive even if the core developers later leave. Core developer participation in the final stages of development also increases the chances of a project being considered successful. Core developers are not important during the early stages of development.

The model presented here aids in gaining a better understanding of the conditions necessary for open source projects to succeed. With further iterations of development, including supplementing the model with better data-based values for parameters and adding additional emergent properties for validation purposes, the model could move further into the realm of prediction. In this case, it would be possible to feed real-life conditions into the model and then observe a given project as it progresses (or lack of progresses) in the FLOSS environment, potentially leading to a better understanding of the FLOSS development process and the conditions necessary for project success.

CHAPTER 8

CONCLUSION

Traditional software engineering is studied in order to understand what methods are worthwhile (e.g., what processes produce software with properties including, but not limited to: high quality, low bug counts, fast development, safety critical, and cost effective). The research presented here is an extension into FLOSS engineering. Namely, many of the activities that occur in FLOSS development are contrary to traditional software engineering best practices, yet in some cases FLOSS manages to produce excellent software. This makes the FLOSS development process worth studying in order to better understand what causes these positive characteristics and how they may be adapted to improve all forms of software engineering.

In general, there is concern about the lack of research using empirical data in software engineering [214], [215], [216]. It is often difficult or impossible to obtain data from proprietary software engineering projects. Even if data is obtainable, it may not be sharable due to non-disclosure agreements [158]. This means the results cannot be independently validated. The work presented here adds to the body of software engineering research that is heavily based on empirical data. In addition, because the FLOSS data used is public, the results can be independently replicated for validation purposes. However, at the same time, it has been shown that extraordinary caution must be exercised when using FLOSS data. While FLOSS is attractive to study because the process naturally captures massive amounts of data, this does not solve the "garbage in, garbage out" problem. This is especially true with human-assigned data, such as project development status, which is prone

to errors. Care must be taken also when dealing with data collected over time; fields may change, as may data collection methods. Other anomalies exist that are data-specific, such as abnormally large commits, which might seem like tremendous progress on a project but in reality are a result of migrating from one SCM system to another. As such, the rich data available from FLOSS development processes, upon closer inspection, actually requires close examination and extensive filtering before it can be used. In some cases the filtering is so severe that the resulting data sets may be insufficient from which to draw conclusions. Hopefully projects like FLOSSmole will continue to mature and improve the quality of the data through enhanced collection techniques. Likewise, newer data collection projects like FLOSSMetrics are able to learn from the shortcomings and problems others have already encountered in order to improve their own data sets. Unfortunately, until higher quality data is available, it is mandatory that those using FLOSS data invest sufficient time in understanding the information before using it. This should lead to reduced rework and more confidence in conclusions derived.

The goal of this research is to better understand the FLOSS development process through the use of public data and agent-based modeling. A particular focus is on understanding why some projects are successful, what causes this success, and how success can be influenced. Through a better understanding of the development process, positive attributes present in FLOSS may be applied to software engineering in general.

In order to study the success of FLOSS projects, it is necessary to first understand what success means in the FLOSS domain. Using modeling, a number of proposed success metrics are explored. In particular, it is found that there is a difference between consumer-oriented and producer-oriented success metrics. The different impact of these two families of metrics on the FLOSS development process is explored. Consumer-oriented metrics re-

sult in a larger spread of projects being worked on while producer-oriented metrics exhibit bandwagon effects. The winner-takes-all dynamics seen when using producer-oriented metrics closely matches the dynamics seen in real world data, with a handful of FLOSS projects being tremendously successful while the majority of projects remain inert. However, within producer-oriented metrics, it is shown that there is a large difference among the proposed metrics. The lack of overlap among the metrics indicates that choice of metrics does matter, and how success is defined may vary by person and scenario. Using multiple metrics to judge success likely will provide a more rounded, stable, and potentially superior indicator when evaluating the state of a FLOSS project.

In order to understand what can be done to increase the chances of success, the approach taken by this research is to understand how consumers and developers choose FLOSS projects. It is assumed that people, and not projects, are a limited resource and that projects must compete for consumers' and developers' attention in order to survive. FLOSSSim indeed performs best when the agents to project ratio is significantly less than 1, supporting this basic assumption and lending credibility to this approach for modeling the FLOSS development process.

Although literature includes many reasons why people become involved in FLOSS, ranging from pure conjecture to survey-based conclusions, this research takes a unique approach by attempting to understand the related concept of how consumers and developers select projects within the FLOSS domain. Beyond simply identifying factors that are important, unique to this research is an attempt to derive the importance of each factor, where the results are based on a model that is backed by publicly available project data. Of the five factors explored for developers, a project's maturity is found to always be important. In addition, the popularity of a project with other developers, the accumulated work on a project,

and the popularity of a project with users are also found to sometimes be important factors in selecting projects. Surprisingly, similarity of interests is not shown to be an important factor. The heterogeneity of the importance of current resources, cumulative resources, and number of downloads indicates that there is diversity in the factors that motivate individuals to choose projects. Much like the heterogeneity found in the reasons people participate in FLOSS, different individuals choose projects for different reasons. For example, those interested in the popularity of projects with developers may be driven by the potential for reputation gain, may be interested in job opportunities, or may simply enjoy sharing and learning new skills that are available from these projects. Those attracted to projects based on the work already completed may be interested in opportunities that are project size dependent, while those that prefer well-used projects may be motivated by recognition or altruism.

Based on the evolved utility weights in FLOSSSim, developers exhibit the following preferences when selecting projects:

- Projects that are popular with developers are more likely to be selected.
- Large projects with a significant portion of work already completed are more likely to be selected.
- Projects that are popular with users are more likely to be selected.
- Projects in lower development stages are more likely to be selected.

Because projects must be selected by developers in order to progress, these findings also provide insight into what conditions increase the probability of success.

Consumers remain a largely unstudied group in the FLOSS domain. To help address this void, both FLOSSSimple and FLOSSSim include consumers for the purpose of better

understanding their impact on the FLOSS development process. Exploring factors that are important to users choosing projects reveals that, unlike developers, a project matching a consumer's interests is important. This is not surprising since consumers are not subject to many of the motivating factors experienced by developers; they simply want software that solves their problems. However, the lack of stability in FLOSSSim's utility weights indicate the selection of projects by consumers is not critical in order for the model to match the empirical data. It appears that consumers' choices of projects have only a minimal influence on the FLOSS development process.

In order for a project to be successful, it must be desirable on multiple fronts; having a single desirable factor is simply not enough to compete with other projects to attract developers. Projects with a single element of appeal have only a small window of opportunity to increase their desirability before developers move on to other projects.

Some projects are inherently successful while other projects are inherently unsuccessful. Inherently successful projects thrive regardless of the surrounding FLOSS landscape. Likewise, inherently unsuccessful projects remain unsuccessful even if the successful projects are eliminated. This behavior, seen in FLOSSSim with the cessation of adding new projects, is consistent with the winner-takes-all behavior observed in FLOSS. That is, there is often a prevailing project satisfying the needs of the community in a category, and other similar projects fail to compete. Most people interested in a topic will simply choose the dominant project rather than invest in a similar underdog project. These dynamics occur because successful projects exhibit a lock-in effect. Once a project exhibits signs of being successful, developers have a tendency to continue working on the project, ignoring the changes in the surrounding FLOSS landscape and helping ensure that the project remains successful into the future. This behavior also indicates there may be a boredom threshold;

projects that are below this static limit largely do not receive contributions and rarely reach critical mass, even when there are no better projects for developers to work on.

A key finding of this research is that FLOSS is a producer-driven process. This is shown consistently throughout the modeling process. From the high evolved producer numbers to the use of producer-oriented success metrics better matching the empirical data to the elimination of consumers from the model resulting in only slightly changed fitness values, all indicators point to developers being the driving force of the FLOSS process. This finding confirms that the ability of a project to attract developers is key to the project's success.

Passive consumers are a distant second to developers in influencing the FLOSS development process, as demonstrated by multiple components of the models. The frequency that consumers use software does not affect the ability of FLOSSSim to match empirical data. If given the ability to select projects using different weights than developers, the consumers' weights are unstable and work over a much larger range without negatively affecting the ability of FLOSSSim to match empirical data. Using consumer-oriented success metrics results in characteristics that do not match that of empirical FLOSS data. However, consumers are not entirely without affect on the development process. Users are able to indirectly influence developers by affecting download counts, which are then used by developers when selecting projects. This minor influence is observed by the slightly decreased fitness values in FLOSSSim when consumers select projects randomly or are eliminated entirely from the model.

Since FLOSS is a producer-driven process, one might expect projects aimed at a developer audience to have an increased probability of success. FLOSSSimple indicates that projects biased towards producers needs tend to outperform projects that are biased

towards consumer needs. FLOSSSim shows in specific circumstances a very weak link between the target audience of a project and the probability of success. These findings are consistent with existing FLOSS literature, which shows that projects aimed at developers are more likely to succeed than projects aimed at end-users. However, it is pointed out that FLOSSSim's performance is strongly tied to the distribution of peoples' interests and that the assumptions used in the model may be incorrect; further investigation into target audiences' effect on FLOSS is therefore warranted.

Core developers are an important component of successful FLOSS. Even more than developers, core developers are a limited resource. Because of this, FLOSSSim is used to explore when it is most critical to receive core developer contributions. Projects that have core developers involved during the middle stages of development have the highest likelihood of being successful, as contributions during this time are able to propel a project through a critical, and often fatal, period. Late core developer involvement, to a lesser extent, also increases the chances of project success. Core developer involvement during the early stages of a project does not in fact increase the probability of success. This is a component of the FLOSS development process that is not very robust. Core developers are a limited resource, and it is important that they be involved during a specific time of development. Chances of not only attracting core developers but timing their involvement during a critical period may be one of the reasons that few projects thrive.

The influence of core developers on the success of a project is particularly apt for companies or individuals looking for a suitable FLOSS solution. Projects in early stages with core developers may appear active and likely to mature, but in reality core developers in the early stages do not increase the chances of project success. A safer choice would be to select a project that is in the mid-stages of development and has a strong group of core

developers, as there is a greater chance this is an up-and-coming project. The presence or absence of core developers in the late stages of development is less relevant for predicting the future of a project because by this time the existing features of the project can provide a good idea of if the project is worthwhile (i.e., functional, well-documented, user-friendly, high quality). In addition, if a project is missing key features, there is a greater chance the functionality will be added in a young project; mature projects are more likely to be in a maintenance and support phase, where new feature requests are not being considered. Therefore, a mid-stage project with core developers may be a better choice over a mature project if additional functionality or enhancements are required by those choosing the software.

Taken together with recommendations that projects not be released into the FLOSS domain until a kernel of working code has already been developed, projects that are initially developed outside of the FLOSS domain that then retain the original developers' involvement as core developers, at least temporarily after being released as FLOSS, have a substantial advantage over other projects. These projects will be significantly more likely to survive the transition to FLOSS and become successful.

From the findings of this research, it can be seen that FLOSS has managed to address and/or avoid some of the problems that plague traditional software engineering. For example, this research demonstrates that FLOSS is a producer-driven process. Passive consumers, those that use open source software but provide no form of contribution to projects, have little effect on projects. Unusual to the FLOSS domain is the fact that most of the developers are also consumers of the software – that is consumers have an itch and also the skills to scratch the itch, making them also developers. This solves the major software engineering problem of requirements elicitation. Because the developers are also the users, the

development team inherently already understands the requirements of the software. Eliminating a requirements elicitation stage removes the misunderstandings and ambiguities that inherently occur during this phase while also increasing efficiency, allowing more time and effort to be spent addressing the problems rather than understanding what problems need to be addressed.

There are many approaches to improving the requirements elicitation process. FLOSS addresses this problem extremely effectively by allowing those with knowledge of the requirements to also write the code. Inherently this doesn't necessarily work; it is quite common for people from all walks of life to possess at least some limited programming skills and thus be able to hack together software solutions to their problems. The programs produced, however, aren't necessarily correct, maintainable, extendable, etc. FLOSS, on the other hand, not only is able to get users who understand the requirements to write the code, but is also able to tap into a large pool of developers, meaning there is a better chance of a skilled developer writing the code, resulting in both good design and correctness. Barring this, the openness of the process means that other developers are able to fix, improve, and refactor the code as necessary. Interestingly enough, by employing this methodology, FLOSS has largely eliminated the thought-to-be-necessary design documents associated with high quality software engineering. This can be seen as an advantage, as many programmers loathe creating design documents (they'd rather be writing code), and building and updating these documents takes time. If as-good or better software can be created without generating official or formal design documents through an improved understanding of requirements, this could be a goal for software engineering processes in general.

The implication of this finding is that getting the end-users more involved in the development process will result in better software, with the users receiving the software they

want, not what the developers think the users need. Requirements elicitation has always been a problem in software engineering. FLOSS has managed to very successfully address the gap between users and developers by making sure the users are also the developers. This results is software that performs the necessary tasks, and performs them well, that the users are most interested in. Traditional software engineering companies cannot address this to the level done by open source. If a customer approaches a software company with a problem, chances are the customer does not possess the necessary skills or resources to write the code themselves. However, the more action a software engineering team takes to include the customer throughout the development process, the better the chance that the resulting software will indeed address the customer's needs.

In addition to being a producer-driven process, the majority of FLOSS developers are actively working on projects almost all the time. This is reflected in both empirical data and the performance of FLOSSSim, where 91% and 93% of developers respectively are actively contributing to a project at any given time. This highlights a high motivation level that may not be present in proprietary software engineering. FLOSS developers are self-motivated to be involved in open source development and may choose the tasks they find interesting; developers are therefore likely excited to work and driven to perform their best. In addition, the high participation level shows the ability of FLOSS to very efficiently utilize limited resources; essentially all developers' skills, regardless of what they may be, are able to be put to use at any given time. The combined strength of motivated developers and efficient distribution of skills has the potential to increase the quality of the software produced and speed at which it is developed. This is something FLOSS does very well and an area that traditional software engineering should strive to improve. Some companies understand that keeping employees interested in their tasks results in better software.

Google, for example, insists that its employees spend 20% of their time working on any project that interests them, so long as it is not their main project. Doing so essentially embraces a component of the FLOSS development process, hopefully resulting in innovative, or at least good, solutions to software problems by highly motivated employees. Incorporating this FLOSS practice has worked well for Google, which claims that 50% of the projects launched in the second half of 2005 were actually a result of the company's 20% time policy [217].

Of all the methods employed in proprietary software engineering, agile software development methodologies exhibit the largest number of parallels to FLOSS development. According to the Manifesto for Agile Software Development [218], customers should be represented throughout the entire development process, unlike more traditional software engineering processes, where customers are typically present during the initial requirements elicitation phase and not much thereafter. FLOSS also emphasizes the importance of users' presence throughout the entire lifecycle of project, noting that their contributions and feedback enhance the software produced. The fact that the developers are also the consumers in FLOSS further ensures customer involvement throughout the development process. Agile methods also rely on small, tightly knit groups of core developers, much like the groups of core developers that increase the chances of FLOSS success. Agile development methodologies encourage short iterative lifecycles, resulting in frequent releases of working software. Likewise, it is recommended that FLOSS projects release early and release often, even if this means releasing incomplete software. Both agile and FLOSS development processes function well with changing requirements, and both development methodologies seem to emphasize producing software over documentation. Agile development also stresses the importance of individuals, noting that developers should be motivated and try-

ing to avoid processes that unnecessarily impede progress. FLOSS ensures developers are motivated because contributions are voluntary. Eliminating cumbersome processes is also important in FLOSS so that projects are able to attract and retain developers. One major difference between agile and FLOSS development is the high value agile methods place on co-location and face-to-face communication. FLOSS, on the other hand, thrives with widely geographically distributed teams and relies heavily on tools (e.g., email, forums) for a communication medium.

The research presented here has focused on collaboratively written software. However, there are other digital public goods that may also benefit from this work. The most obvious example is Wikipedia, which has many parallels to FLOSS. In the case of Wikipedia, instead of software, articles are the product that is created. Like FLOSS, contributions to Wikipedia come from volunteers with different motivations to participate. The distribution of contributions is also skewed, with 90% of the users contributing fewer than 10% of the edits [219]; the number of people per article and the number of edits per article follow a power law distribution [220]. Also like FLOSS, there exists a continuum of fringe to core contributors that are afforded different levels of control over articles. Like the FLOSS domain, which includes hundreds of thousands of projects, the number of articles in Wikipedia is also huge, consisting of 73 million articles in 281 languages with 1.2 billion edits by 30 million users [221]. Significantly younger than the FLOSS movement, the online encyclopedia was created in January, 2001 [222]. Unlike FLOSS, the rate of article creation is dropping off, perhaps indicating that Wikipedia articles have a shorter development lifespan compared to FLOSS projects, but not necessarily a shorter lifespan period, as there is no evidence that Wikipedia as a source of information is becoming obsolete. Rather, many Wikipedia articles seem to have entered a state similar to the maintenance phase in software

development, where the quality of the articles continues to increase, even if the bulk of the work is already complete. The motivating factors for participating in Wikipedia may be similar to FLOSS and include reputation and commitment to group identity [223]. Finally, success is also ambiguous for Wikipedia articles, although this may relate to article quality. Quality is suggested to be associated with the number of edits and unique editors to an article [224]. Quality might also relate to factual accuracy [225] and credibility [226]. Like with FLOSS, article creation and editing leaves a digital data trail relating to the contributors while information on consumers is not readily available. With so many similarities between FLOSS and Wikipedia, the possibility of easily extending this research to explore Wikipedia appears promising.

Although not exactly digital public goods, other digital phenomena such as the blogosphere, YouTube, etc. also include components of crowdsourcing. Therefore, this research might also be extended to these domains as well.

In conclusion, this research uses empirical data and agent-based modeling to gain a better understanding of the FLOSS domain. The model created is able to reproduce key characteristics of the FLOSS development process and may even be used for prediction purposes. Through the use of the model, a better understanding of FLOSS has been gained, including what it means to be successful in the FLOSS domain. Factors that affect the success of projects have also been identified, along with their importance. The positive aspects of FLOSS combined with the findings of this research can be used to improve software engineering in general. Finally, while the focus of this research has been FLOSS, with relatively minor modifications it may be possible to study other similar digital public goods.

REFERENCES

[1] R. A. Ghosh, B. Krieger, R. Glott, and G. Robles, "Part 4: Survey of developers," in *Free/Libre and Open Source Software: Survey and Study*. Maastricht, The Netherlands: University of Maastricht, The Netherlands, Jun. 2002.

[2] (2011, Jul.) History of the OSI. Open Source Initiative. San Francisco, CA, USA. [Online]. Available: http://www.opensource.org/history

[3] (2011, Jul.) The open source definition. Open Source Initiative. San Francisco, CA, USA. [Online]. Available: http://www.opensource.org/docs/osd

[4] (2011, Jul.) Open source licenses. Open Source Initiative. San Francisco, CA, USA. [Online]. Available: http://www.opensource.org/licenses/index.html

[5] (2011, Jul. 13) Why "free software" is better than "open source". Free Software Foundation. Boston, MA, USA. [Online]. Available: http://www.gnu.org/philosophy/free-software-for-freedom.html

[6] (2010, Oct. 5) The free software definition. Free Software Foundation. Boston, MA, USA. [Online]. Available: http://www.gnu.org/philosophy/free-sw.html

[7] R. Stallman. (2011, Jul. 13) Why open source misses the point of free software. Free Software Foundation. Boston, MA, USA. [Online]. Available: http://www.gnu.org/philosophy/open-source-misses-the-point.html

[8] "Linux kernel software quality and security better than most proprietary enterprise software, 4-year Coverity analysis finds." Press release, 2004, Coverity Incorporated.

[9] "Analysis of the Linux kernel," Research report, 2004, Coverity Incorporated.

[10] B. Chelf, "Measuring software quality: A study of open source software," Research report, 2006, Coverity Incorporated.

[11] R. Lemos, "Security research suggests Linux has fewer flaws," *CNET News*, Dec. 13, 2004. [Online]. Available: http://news.cnet.com/Security-research-suggests-Linux-has-fewer-flaws/2100-1002_3-5489804.html

[12] B. Kogut and A. Metiu, "Open-source software development and distributed innovation," *Oxford Review of Economic Policy*, vol. 17, no. 2, pp. 248–264, Summer 2001.

[13] E. S. Raymond, “The cathedral and the bazaar,” Thyrsus Enterprises, Tech. Rep. 3.0, Sep. 11, 2000.

[14] V. Valloppillil, “Halloween document I: Open source software: A (new?) development methodology,” Microsoft, Memorandum Version 1.17, Aug. 11, 1998. [Online]. Available: http://catb.org/~esr/halloween/halloween1.html

[15] M. A. Rossi, “Decoding the “Free/Open Source (F/OSS) software puzzle” a survey of theoretical and empirical contributions,” Dipartimento di Economia Politica, Università degli Studi di Siena, Quaderni 424, Apr. 2004.

[16] C. M. Schweik, R. C. English, M. Kitsing, and S. Haire, “Brooks’ versus Linus’ law: an empirical test of open source projects,” in *DG.O ’08: Proceedings of the 2008 International Conference on Digital Government Research*, S. A. Chun, M. Janssen, and J. R. Gil-García, Eds. Digital Government Society of North America, May 18–21, 2008, pp. 423–424.

[17] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA, USA: Addison-Wesley, 1975.

[18] R. M. Isaac, J. M. Walker, and A. W. Williams, “Group size and the voluntary provision of public goods: Experimental evidence utilizing large groups,” *Journal of Public Economics*, vol. 54, no. 1, pp. 1–36, May 1994.

[19] V. Valloppillil and J. Cohen, “Halloween document II: Linux OS competitive analysis: The next Java VM?” Microsoft, Memorandum Version 1.7, Aug. 11, 1998. [Online]. Available: http://catb.org/~esr/halloween/halloween2.html

[20] (2011, Jul.) Usage statistics and market share of Unix for websites. Q-Success. Maria Enzersdorf, Austria. [Online]. Available: http://w3techs.com/technologies/details/os-unix/all/all

[21] (2011, Jul.) Web server survey. Netcraft. Bath, England. [Online]. Available: http://news.netcraft.com/archives/category/web-server-survey/

[22] (2011, Jul.) Usage of web servers for websites. Q-Success. Maria Enzersdorf, Austria. [Online]. Available: http://w3techs.com/technologies/overview/web_server/all

[23] J. Cownie. (2011, Jul. 8,) Most reliable hosting company sites in June 2011. Netcraft. Bath, England. [Online]. Available: http://news.netcraft.com/archives/2011/07/08/most-reliable-hosting-company-sites-in-june-2011.html

[24] (2010, Dec. 24,) OpenDocument adoption. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/OpenDocument_adoption

[25] (2008, Feb. 8,) History of OpenDocument. OASIS. [Online]. Available: http://opendocument.xml.org/milestones

[26] (2000, Jul. 19,) Welcome to the OpenOffice.org source project. OpenOffice.org Foundation. [Online]. Available: http://www.openoffice.org

[27] (2011, Jun.) Browser market share. Net Applications. Aliso Viejo, CA, USA. [Online]. Available: http://marketshare.hitslink.com/report.aspx?qprid=0&qpcal=1&qptimeframe=M&qpsp=149&qpnp=1

[28] (2011, Jun. 11,) Top 5 browsers on Jun 11. StatCounter. [Online]. Available: http://gs.statcounter.com/#browser-ww-monthly-201106-201106-bar

[29] (2011, Jun.) Web browser market share. W3Counter. [Online]. Available: http://www.w3counter.com/globalstats.php?year=2011&month=6

[30] E. Zachte. (2011, Jul. 15) Wikimedia traffic analysis report - browsers e.a. Wikimedia. [Online]. Available: http://stats.wikimedia.org/wikimedia/squids/SquidReportClients.htm

[31] G. Sisson. (2010, Oct.) DNS survey: October 2010. The Measurement Factory. Westminster, CO, USA. [Online]. Available: http://dns.measurement-factory.com/surveys/201010/

[32] R. A. Ghosh, "Advancing the research agenda on free / open source software," workshop report Free/Libre and Open Source Software: Survey and Study. Brussels: European Commission, Oct. 14, 2002. [Online]. Available: http://www.flossproject.org/report/workshopreport.htm

[33] T. Wichmann, "Part 2: Firms' open source activities: Motivations and policy implications," FLOSS Final Report: Free/Libre Open Source Software: Survey and Study. Berlin, Germany: Berlecon Research, Jul. 2002. [Online]. Available: http://www.berlecon.de/studien/downloads/200207FLOSS_Activities.pdf

[34] D. Tapscott and A. D. Williams, *Wikinomics: How Mass Collaboration Changes Everything*. New York, NY, USA: Penguin Group (US) Inc., 2006.

[35] A. Watson, "Reputation in open source software," Northeastern University, Boston, MA, USA, working paper 1.0, Sep. 2005. [Online]. Available: http://opensource.mit.edu/papers/watson.pdf

[36] P. Wayner, *Free for All: How Linux and the Free Software Movement Undercut the High-Tech Titans*. New York, NY, USA: HarperInformation, 2000.

[37] P. Wagstrom, J. Herbsleb, and K. Carley, "A social network approach to free/open source software simulation," in *First International Conference on Open Source Systems*, M. Scotto and G. Succi, Eds., Genova, Italy, Jul. 11–15, 2005, pp. 16–23.

[38] T. Smith, "Open source: Enterprise ready – with qualifiers," theOpenEnterprise, Oct. 1, 2002, http://www.theopenenterprise.com/story/TOE20020926S0002. [Online]. Available: http://www.theopenenterprise.com/story/TOE20020926S0002

[39] R. Ghosh, K. Haaland, and B. H. Hall, "Which firms participate in open source software development? A study using data from Debian," paper presented at DIME–DRUID Fundamental on Open and Proprietary Innovation Regimes, Copenhagen, Denmark, May 17, 2008.

[40] A. Bonaccorsi and C. Rossi, "Contributing to the common pool resources in open source software. A comparison between individuals and firms," Institute for Informatics and Telematics, Pisa, Italy, working paper 2nd draft, 2003. [Online]. Available: http://opensource.mit.edu/papers/bnaccorsirossidevelopers.pdf

[41] S. Weber, *The Success of Open Source*. Cambridge, MA, USA: Harvard University Press, 2004.

[42] I. P. Antoniades, I. Samoladas, I. Stamelos, L. Angelis, and G. L. Bleris, "Dynamical simulation models of the open source development process," in *Free/Open Source Software Development*, S. Koch, Ed. Hershey, PA, USA: Idea Group, Incorporated, 2005, pp. 174–202.

[43] Y. Mao, J. Vassileva, and W. Grassmann, "A system dynamics approach to study virtual communities," in *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2007, p. 178a.

[44] E. Katsamakas and N. Georgantzas, "Why most open source development projects do not succeed?" in *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. Washington, DC, USA: IEEE Computer Society, 2007, p. 3.

[45] J.-M. Dalle and P. A. David, "SimCode: Agent-based simulation modelling of open-source software development," EconWPA, Industrial Organization, Nov. 1, 2004.

[46] W. Scacchi, J. Feller, B. Fitzgerald, S. A. Hissam, and K. Lakhani, "Understanding free/open source software development processes," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 95–105, 2006.

[47] K. Kowalczykiewicz, "Libre projects lifetime profiles analysis," in *Free and Open Source Software Developers' European Meeting 2005*, Brussels, Belgium, 2005.

[48] Y. Wang, "Prediction of success in open source software development," Master of science dissertation, University of California, Davis, Davis, CA, Spring 2007.

[49] Y. Gao, Y. Huang, and G. Mady, "Data mining project history in open source software communities," Presented at NAACSOS 2004. Pittsburg, PA, USA: North American Association for Computational Social and Organization Sciences, Jun. 2004.

[50] I. P. Antoniades, I. Stamelos, L. Angelis, and G. L. Bleris, "A novel simulation model for the development process of open source software projects," *Software Process: Improvement and Practice*, vol. 7, no. 3–4, pp. 173–188, 2002.

[51] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the Apache server," in *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*. New York, NY, USA: ACM, 2000, pp. 263–272.

[52] J. Bitzer and P. J. Schröder, "Bug-fixing and code-writing: The private provision of open source software," *Information Economics and Policy*, vol. 17, no. 3, pp. 389–406, Jul. 2005.

[53] M. Bilodeau and A. Slivinski, "Toilet cleaning and department chairing: Volunteering a public service," *Journal of Public Economics*, vol. 59, no. 2, pp. 299–308, 1996.

[54] R. A. Ghosh and P. A. David, "The nature and composition of the Linux kernel developer community: A dynamic analysis," Stanford Institute of Economic Policy Research, working paper, Mar. 5, 2003.

[55] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*. New York: Oxford University Press, 1993.

[56] G. Madey, Y. Gao, V. Freeh, R. Tynan, and C. Hoffman, "Agent-based modeling and simulation of collaborative social networks," AMCIS 2003, Tampa, FL, USA, Aug. 2003.

[57] J. Xu, Y. Gao, S. Christley, and G. Madey, "A topological analysis of the open source software development community," in *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*. Hawaii, USA: Society Press, 2005, pp. 4–7.

[58] Y. Gao, G. Madey, and V. Freeh, "Modeling and simulation of the open source software community," in *Agent-Directed Simulation Symposium, SpringSim'05*. San Diego, CA: Society for Modeling and Simulation International, Apr. 2005, pp. 113–122.

[59] Y. Gao and G. Madey, "Towards understanding: a study of the SourceForge.net community using modeling and simulation," in *SpringSim '07: Proceedings of the 2007 Spring Simulation Multiconference*. Norfolk, VA, USA: Society for Computer Simulation International, 2007, pp. 145–150.

[60] N. Smith, A. Capiluppi, and J. Fernández-Ramil, "Users and developers: An agent-based simulation of open source software evolution," in *SPW/ProSim*, ser. Lecture Notes in Computer Science, Q. Wang, D. Pfahl, D. M. Raffo, and P. Wernick, Eds., vol. 3966. Shanghai, China: Springer, May 20–21, 2006, pp. 286–293.

[61] W. Scacchi, "Understanding open source software evolution," in *Software Evolution and Feedback: Theory and Practice*, N. H. Madhavji, J. C. Fernández-Ramil, and D. E. Perry, Eds. New York, NY, USA: John Wiley & Sons, Ltd, 2006, ch. 9, pp. 181–205.

[62] C. Jensen and W. Scacchi, "Process modeling across the web information infrastructure," *Software Process: Improvement and Practice*, vol. 10, no. 3, pp. 255–272, 2005.

[63] A. Borshchev and A. Filippov, "From system dynamics and discrete event to practical agent based modeling: Reasons, techniques, tools," Proceedings of the 22nd International Conference of the System Dynamics Society, M. Kennedy, G. W. Winch, R. S. Langer, J. I. Rowe, and J. M. Yanni, Eds. Oxford, England, UK: System Dynamics Society, Jul. 25–29, 2004.

[64] K. J. Stewart, A. P. Ammeter, and L. M. Maruping, "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects," *Information Systems Research*, vol. 17, no. 2, pp. 126–144, Jun. 2006.

[65] W. Scacchi, "Understanding the requirements for developing open source software systems," *IEE Proceedings - Software*, vol. 149, no. 1, pp. 24–39, 2002.

[66] K. Crowston and B. Scozzi, "Open source software projects as virtual organizations: Competency rallying for software development," in *IEE Proceedings Software*, vol. 149, no. 1, 2002, pp. 3–17.

[67] K. J. Stewart and A. P. Ammeter, "An exploratory study of factors influencing the level of vitality and popularity of open source projects," in *Proceedings of the 23rd International Conference on Information Systems*, L. Applegate, R. Galliers, and J. DeGross, Eds., Barcelona, Spain, Dec. 15–18, 2002, pp. 853–857.

[68] (2010) About Netcraft. Online. Netcraft. Bath, England. Accessed: Nov. 14, 2010. [Online]. Available: http://news.netcraft.com/about-netcraft/

[69] D. Weiss, "Measuring success of open source projects using Web search engines," in *Proceedings of the First International Conference on Open Source Systems (OSS 2005)*, M. Scotto and G. Succi, Eds., Genova, Italy, 2005, pp. 93–99.

[70] K. Crowston, J. Howison, and H. Annabi, "Information systems success in free and open source software development: Theory and measures," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 123–148, March/April 2006.

[71] K. J. Stewart and S. Gosain, "The moderating role of development stage in free/open source software project performance," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 177–191, 2006.

[72] R. English and C. M. Schweik, "Identifying success and tragedy of FLOSS commons: A preliminary classification of SourceForge.net projects," in *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. Washington, DC, USA: IEEE Computer Society, 2007, p. 11.

[73] A. Capiluppi and M. Michlmayr, "From the cathedral to the bazaar: An empirical study of the lifecycle of volunteer community projects," in *Open Source Development, Adoption and Innovation*, J. Feller, B. Fitzgerald, W. Scacchi, and A. Silitti, Eds., International Federation for Information Processing. Limerick, Ireland: Springer, Jun. 11–14, 2007, pp. 31–44.

[74] A. Capiluppi, P. Lago, and M. Morisio, "Evidences in the evolution of OS projects through changelog analyses," Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, Eds., Portland, OR, USA, May 3–11, 2003, pp. 19–24.

[75] S. Koch, "Exploring the effects of SourceForge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis," *Empirical Software Engineering*, vol. 14, no. 4, pp. 397–417, 2009.

[76] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

[77] R. A. Ghosh and V. V. Prakash, "The Orbiten free software survey," *First Monday*, vol. 5, no. 7, Jul. 3, 2000. [Online]. Available: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/769/678

[78] W. Maass, "Inside an open source software community: Empirical analysis on individual and group level," in *Collaboration, Conflict and Control: Proceedings of the 4th Workshop on Open Source Software Engineering*, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, Eds., Edinburgh, Scotland, May 25, 2004, pp. 65–70.

[79] S. Koch and G. Schneider, "Effort, co-operation and co-ordination in an open source software project: GNOME," *Information Systems Journal*, vol. 12, no. 1, pp. 27–42, 2002.

[80] S. Krishnamurthy, "Cave or community?: An empirical examination of 100 mature open source projects," *First Monday*, vol. 7, no. 6, Jun. 2002. [Online]. Available: http://www.firstmonday.org/issues/issue7_6/krishnamurthy/index.html

[81] M. Michlmayr, "Software process maturity and the success of free software projects," in *Software Engineering: Evolution and Emerging Technologies*, K. Zielinski and T. Szmuc, Eds. Krakow, Poland: IOS Press, 2005, pp. 3–14.

[82] J. Howison and K. Crowston, "The perils and pitfalls of mining SourceForge," in *Proceedings of the Mining Software Repositories Workshop at the International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, 2004, pp. 7–11.

[83] B. Allombert. (2010, Nov. 14,) Debian popularity contest. Online. Debian. [Online]. Available: http://popcon.debian.org

[84] R. Turk. (2007, Jul.) SourceForge stats demystified. Blog. SourceForge. [Online]. Available: http://sourceforge.net/blog/sourceforge-stats-demystified/

[85] (2010, Jul.) How do you measure a project's vitality? freshmeat. [Online]. Available: http://help.freshmeat.net/faqs/statistics/how-do-you-measure-a-projects-vitality

[86] E. Ostrom, R. Gardner, and J. Walker, *Rules, Games and Common Pool Resources*. Ann Arbor, MI: University of Michigan Press, 1994.

[87] T. H. Jerdee and B. Rosen, "Effects of opportunity to communicate and visibility of individual decisions on behavior in the common interest," *Journal of Applied Psychology*, vol. 59, no. 6, pp. 712–716, 1974.

[88] H. Tajfel, *Human Groups and Social Categories: Studies in Social Psychology*. Cambridge, UK: Cambridge University Press, 1981.

[89] R. Axelrod, *The Evolution of Cooperation*. New York: Basic Books, 1984.

[90] J. Fox and M. Guyer, "Group size and others' strategy in an n-person game," *Journal of Conflict Resolution*, vol. 21, no. 2, pp. 323–338, Jun. 1977.

[91] P. Kolloc and M. A. Smith, "Communities in cyberspace," in *Communities in Cyberspace*, P. Kolloc and M. A. Smith, Eds. London: Routledge, 1999, ch. 1, pp. 3–28.

[92] K. Kuwabara, "Linux: A bazaar at the edge of chaos," *First Monday*, vol. 5, no. 3, Mar. 6, 2000. [Online]. Available: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/731/640

[93] J. Lerner and J. Tirole, "Some simple economics of open source," *The Journal of Industrial Economics*, vol. 50, no. 2, pp. 197–234, Jun. 2002.

[94] N. P. Radtke and M. A. Janssen, "Consumption and production of digital public goods: Modeling the impact of different success metrics in open source software development," *International Journal of Intelligent Control and Systems*, vol. 14, no. 1, Mar. 2009.

[95] R. van Wendel de Joode, "Understanding open source communities: An organizational perspective," Ph.D. dissertation, Technische Universiteit Delft, The Netherlands, Sep. 26, 2005.

[96] M. Olson, *The Logic of Collective Action: Public Goods and the Theory of Groups*, ser. Harvard Economic Studies. Cambridge, MA, USA: Harvard University Press, 1965, vol. 124.

[97] G. Hardin, "The tragedy of the commons," *Science*, vol. 162, no. 3859, pp. 1243–1247, 1968.

[98] K. R. Lakhani, B. Wolf, and J. Bates, "The Boston Consulting Group hacker survey, release 0.3," Online, 2002, http://flosscom.net/index.php?option=com_docman\&task=doc_view\&gid=45.

[99] C. M. Schweik, "An institutional analysis approach to studying libre software 'commons'," *Upgrade: The European Journal for the Informatics Professional*, vol. VI, no. 3, pp. 17–27, Jun. 2005.

[100] S. C. Smith and A. Sidorova, "Survival of open-source projects: A population ecology perspective," in *ICIS 2003. Proceedings of International Conference on Information Systems 2003*, Seattle, WA, 2003.

[101] J. Lerner and J. Tirole, "The scope of open source licensing," *Journal of Law, Economics, and Organization*, vol. 21, no. 1, pp. 20–56, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1093/jleo/ewi002

[102] (2006) Report of license proliferation committee and draft FAQ. Online. Open Source Initiative. San Francisco, CA, USA. [Online]. Available: http://opensource.org/proliferation-report

[103] (2010, Apr. 27,) Licenses. Online. Free Software Foundation. Boston, MA, USA. [Online]. Available: http://www.gnu.org/licenses/

[104] C. Fershtman and N. Gandal, "The determinants of output per contributor in open source projects: An empirical examination," Centre for Economic Policy Research, London, CEPR Discussion Paper 4329, Mar. 2004. [Online]. Available: http://www.cepr.org/pubs/dps/DP4329.asp

[105] E. Katsamakas, B. Janamanchi, W. Raghupathi, and W. Gao, "A classification analysis of the success of open source health information technology projects," *International Journal of Healthcare Information Systems and Informatics*, vol. 4, no. 4, pp. 19–36, 2009.

[106] S. Koch, "Measuring the efficiency of free and open source software projects using data envelopment analysis," in *Emerging Free and Open Source Software Practices*, S. K. Sowe, I. G. Stamelos, and I. Samoladas, Eds. Hershey, PA, USA: IGI Global, 2008, pp. 25–46.

[107] A. I. Wasserman and E. Capra, "Evaluating software engineering processes in commercial and community open source projects," in *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. Washington, DC, USA: IEEE Computer Society, May 20–26, 2007, p. 1.

[108] N. Jullien and J.-B. Zimmermann, "New approaches to intellectual property: From open software to knowledge-based industrial activities," in *International Handbook on Industrial Policy*, P. Bianchi and S. Labory, Eds. Cheltenham, UK: Edward Elgar Publishing Limited, 2006, ch. 12, pp. 243–264.

[109] C. M. Schweik and R. English, "Tragedy in the FOSS commons? Investigating the institutional designs of free/libre and open source software projects," *First Monday*, vol. 12, no. 2, Feb. 5, 2007. [Online]. Available: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1619/1534

[110] S. Vaughan-Nichols, "Leaked memo revives SCO-Microsoft connection furor," *eWeek*, Mar. 4, 2004.

[111] "Jury verdict: The SCO Group, Inc. vs. Novell Inc. case no. 2:04-CV-139 TS," United States Court for the District of Utah, Mar. 30, 2010. [Online]. Available: http://www.groklaw.net/pdf2/Novell-846.pdf

[112] E. Montalbano, "Novell won't persue Unix copyrights," *PCWorld*, Aug. 15, 2007.

[113] W. Oh and S. Jeon, "Membership herding and network stability in the open source community: The Ising perspective," *Management Science*, vol. 53, no. 7, pp. 1086–1101, Jul. 2007.

[114] S. Vaughan-Nichols, "Commercializing open-source stirs debate," *eWeek*, Aug. 4, 2005. [Online]. Available: http://www.eweek.com/c/a/Linux-and-Open-Source/Commercializing-OpenSource-Stirs-Debate/

[115] A. Bonaccorsi, D. Lorenzi, M. Merito, and C. Rossi, "Business firms' engagement in community projects. Empirical evidence and further developments of the research," FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development. Washington, DC, USA: IEEE Computer Society, 2007.

[116] M. Levesque, "Fundamental issues with open source software development," *First Monday*, vol. 9, no. 4, Apr. 5, 2004. [Online]. Available: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1137/1057

[117] D. Weiss, "Quantitative analysis of open source projects on SourceForge," in *Proceedings of the First International Conference on Open Source Systems (OSS 2005)*, M. Scotto and G. Succi, Eds., Genova, Italy, 2005, pp. 140–147.

[118] N. Xu, "An exploratory study of open source software based on public project archives," Master's thesis, Concordia University, Montreal, QC, Canada, Spring 2003.

[119] E. S. Raymond, "Homesteading the noosphere," *First Monday*, vol. 3, no. 10, Oct. 5, 1998. [Online]. Available: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/621/542

[120] J. Holck and N. Jørgensen, "Do not check in on red: Control meets anarchy in two open source projects," in *Free/Open Source Software Development*, S. Koch, Ed. Hershey, PA, USA: Idea Group Inc., 2005, ch. 1, pp. 1–26.

[121] S. O'Mahony, "Non-profit foundations and their role in community-firm software collaboration," Making Sense of the Bazaar: Perspectives on Open Source and Free Software, J. Feller, B. Fitzgerald, S. A. Hissam, and K. R. Lakhani, Eds. Sebastopol, CA, USA: O'Reilly & Associates Publications, 2003.

[122] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability maturity model, version 1.1," *IEEE Software*, vol. 10, no. 4, pp. 18–27, 1993.

[123] N. Jørgensen, "Putting it all in the trunk: Incremental software development in the FreeBSD open source project," *Information Systems Journal*, vol. 11, no. 4, pp. 321–336, 2001.

[124] R. Charles, "Program and programmers evaluation using an annotation model: A case of open source systems," presented at the 2nd FLOSS International Workshop on Free/Libre Open Source Software: Open Models and Cooperation within Online Communities, Rennes, France, Jun. 26–27, 2008. [Online]. Available: http://perso.univ-rennes1.fr/eric.darmon/floss/papers/CHARLES.pdf

[125] S.-Y. T. Lee, H.-W. Kim, and S. Gupta, "Measuring open source software success," *Omega*, vol. 37, no. 2, pp. 426–438, 2009.

[126] "CMMI for development, version 1.2," Online, Carnegie Mellon Software Engineering Institute, Pittsburg, PA, USA, Technical Report 1.2, Aug. 2006. [Online]. Available: http://www.sei.cmu.edu/reports/06tr008.pdf

[127] F. Heylighen, "Why is open access development so successful? Stigmergic organization and the economics of information," Open Source Jahrbuch 2007, B. Lutterbeck, R. Gehring, and M. Bärwolf, Eds. Lehmanns Media, 2007, pp. 165–180.

[128] S. Whang, "Market provision of custom software: Learning effects and low balling," *Management Science*, vol. 41, no. 8, pp. 1343–1353+, 1995.

[129] J. Feller and B. Fitzgerald, *Understanding Open Source Software Development*. Great Britain: Pearson Education Limited, 2002.

[130] G. Hertel, S. Niedner, and S. Herrmann, "Motivation of software developers in open source projects: An Internet-based survey of contributors to the Linux kernel," *Research Policy*, vol. 32, no. 7, pp. 1159–1177, 2003.

[131] A. Hars and S. Ou, "Working for free? - Motivations of participating in open source projects," in *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 7*. Washington, DC, USA: IEEE Computer Society, 2001, p. 7014.

[132] M. C. Paulk, "Practices of high maturity organizations." Atlanta, Georgia, USA: SEPG Conference, Mar. 8–11, 1999.

[133] K. Crowston, H. Annabi, and J. Howison, "Defining open source software project success," in *Proceedings of International Conference on Information Systems (ICIS 2003)*, Seattle, WA, Dec. 2003. [Online]. Available: http://citeseer.ist.psu.edu/article/crowston03defining.html

[134] (2011, Jan. 22,) [chrome] index of /releases. SVN repository. The Chromium Projects. [Online]. Available: http://src.chromium.org/viewvc/chrome/releases/

[135] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*. Orlando, FL, USA: ACM, 2002, pp. 76–85.

[136] B. A. Huberman, D. M. Romero, and F. Wu, "Crowdsourcing, attention and productivity," *Journal of Information Science*, vol. 35, no. 6, pp. 758–765, Dec. 2009.

[137] B. A. Huberman, C. Loch, and A. Önçüler, "Status as a valued resource," *Social Psychology Quarterly*, vol. 67, no. 1, pp. 103–114, 2004.

[138] J. Lampel and A. Bhalla, "The role of status-seeking in online communities: Giving the gift of experience," *Journal of Computer-Mediated Communication*, vol. 12, no. 2, 2007.

[139] L. Tang, "Harnessing the power of social media," presented at SCIDSE Graduate Student Forum Seminar. Arizona State University, Feb. 25, 2010.

[140] K. R. Lakhani and R. G. Wolf, "Why hackers do what they do: Understanding motivation and effort in free/open source software projects," in *Perspectives on Free and Open Source Software*, J. Feller, B. Fitzgerald, S. A. Hissam, and K. R. Lakhani, Eds. Cambridge, Mass.: MIT Press, 2005.

[141] S. E. Ante, "Big blue's big bet on free software," *Business Week*, pp. 78–79, Dec. 10, 2001. [Online]. Available: http://www.businessweek.com/magazine/content/01_50/b3761094.htm

[142] (2009, Dec.) Forge (software). Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Forge_(software)

[143] (2011, Aug.) Sitemap. SourceForge. [Online]. Available: http://sourceforge.net/sitemap.xml

[144] (2011, Aug.) Software search. SourceForge. [Online]. Available: http://sourceforge.net/search/

[145] (2010, Feb.) Project statistics. Wiki. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/Project%20statistics

[146] (2009, Dec.) OSS research. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/OSS%20Research

[147] (2010, Jan.) Open source software used at SourceForge.net. Wiki. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/Open%20Source%20software%20used%20at%20SourceForge.net

[148] (2010, Jan.) Open source software released by SourceForge. Wiki. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/Open%20Source%20software%20released%20by%20SourceForge

[149] G. Madey, Personal communication at SpringSim'09. San Diego, CA, USA: The Society for Modeling and Simulation International, Mar. 22–27, 2009.

[150] (2006, Apr.) Document D04: SourceForge.net: Statistics. [Online]. Available: http://sourceforge.net/docman/display_doc.php?docid=14040&group_id=1

[151] G. Madey. (2009, Dec.) The SourceForge Research Data Archive (SRDA). University of Notre Dame. [Online]. Available: http://srda.cse.nd.edu

[152] Y. Gao, M. Van Antwerp, S. Christley, and G. Madey, "A research collaboratory for open source software research," in *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*. Washington, DC, USA: IEEE Computer Society, 2007, p. 124.

[153] M. Van Antwerp and G. Madey, "Advances in the SourceForge Research Data Archive (SRDA)," in *Fourth International Conference on Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*, Milan, Italy, Sep. 2008.

[154] (2009, Dec.) FAQ. University of Notre Dame. [Online]. Available: http://zerlot.cse.nd.edu/mediawiki/index.php?title=FAQ

[155] (2009, Dec.) SourceForge.net database sublicense terms and conditions. OSTG, Inc. [Online]. Available: http://www.nd.edu/~oss/Data/Sublicense5.pdf

[156] J. Howison, M. Conklin, and K. Crowston, "FLOSSmole: A collaborative repository for FLOSS research data and analyses," *International Journal of Information Technology and Web Engineering*, vol. 1, pp. 17–26, Jul. 2006.

[157] (2009, Dec.) FLOSSmole: Collaborative collection and analysis of free/libre/open source project data. [Online]. Available: http://flossmole.org

[158] *FLOSSMETRICS Description of Work*, 3.0-public ed., Oct. 2008. [Online]. Available: http://flossmetrics.org/docs/DoW-3.0-public.pdf

[159] I. Herraiz, D. Izquierdo-Cortazar, and F. Rivas-Hernández, "FLOSSMetrics: Free/libre/open source software metrics," in *CSMR*, A. Winter, R. Ferenc, and J. Knodel, Eds. IEEE, Mar. 24–27, 2009, pp. 281–284.

[160] (2009, Dec.) Melquiades. [Online]. Available: http://melquiades.flossmetrics.org

[161] S. Dueñes, *Database*, 2nd ed., Nov. 2009. [Online]. Available: http://flossmetrics.org/sections/deliverables/docs/deliverables/WP3/D3.2.-Database.pdf

[162] (2009, Dec.) FLOSSMetrics project. [Online]. Available: http://flossmetrics.org

[163] K. Crowston and M. Squire. (2009, Nov.) [ossmole-discuss] fwd: FLOSSmole license. Mailing list. FLOSSmole. [Online]. Available: http://sourceforge.net/mailarchive/forum.php?thread_name=725EE0CA-D5C2-4D6E-B75B-D8C205452248%40syr.edu&forum_name=ossmole-discuss

[164] *OSSmole Tools Documentation*, 1.0.5 ed., User Manual, FLOSSmole, Nov. 2006. [Online]. Available: http://sourceforge.net/projects/ossmole/files/sfSpiderCode/

[165] C. Garcia-Campos, *The CVSAnalY Manual*, 2.0.0 ed., User Manual, LibreSoft, Apr. 2009. [Online]. Available: http://gsyc.es/~carlosgc/files/cvsanaly.pdf

[166] I. Herraiz, *mlstats man page*, 0.3.3 ed., User Manual, LibreSoft, Jun. 2007. [Online]. Available: https://svn.forge.morfeo-project.org/svn/libresoft-tools/mailinglistst at/trunk/man/mlstats.1

[167] D. I. Cortazar and F. Rivas, *Bicho User Manual v1*, 0.3 ed., User Manual, LibreSoft, Feb. 2009. [Online]. Available: https://svn.forge.morfeo-project.org/svn/libresoft-tools/bicho/trunk/doc/UserManual.txt

[168] J. I. Maletic and A. Marcus, "Data cleansing: Beyond integrity analysis," in *Fifth Conference on Information Quality (IQ 2000)*. Boston, MA: MIT, 2000, pp. 200–209.

[169] K. Orr, "Data quality and systems theory," *Communications of the ACM*, vol. 41, no. 2, pp. 66–71, 1998.

[170] T. C. Redman, "The impact of poor data quality on the typical enterprise," *Communications of the ACM*, vol. 41, no. 2, pp. 79–82, 1998.

[171] E. Rahm and H. H. Do, "Data cleaning: Problems and current approaches," *IEEE Data Engineering Bulletin*, vol. 23, no. 4, pp. 3–13, 2000.

[172] H. Galhardas, D. Florescu, D. Shasha, and E. Simon, "An extensible framework for data cleaning," in *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*. Washington, DC: IEEE Computer Society, 2000, p. 312.

[173] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[174] C. A. Knoblock, K. Lerman, S. Minton, and I. Muslea, "Accurately and reliably extracting data from the web: A machine learning approach," in *Intelligent Exploration of the Web*, P. S. Szczepaniak, J. Segovia, J. Kacprzyk, and L. A. Zadeh, Eds. Heidelberg, Germany: Physica-Verlag GmbH, 2003, pp. 275–287.

[175] (2006, Jun.) Historical rankings. SourceForge. [Online]. Available: http://sourceforge.net/docman/#historical_rankings

[176] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion: For Subversion 1.6: (Compiled from r3658)*, User Manual, 2009. [Online]. Available: http://svnbook.red-bean.com/nightly/en/svn-book.pdf

[177] (2010, Jan.) Git. Wiki. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/Git

[178] (2010, Jan.) Mercurial. Wiki. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/Mercurial

[179] (2010, Jan.) Bazaar. Wiki. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/wiki/Bazaar

[180] (2009, Jul.) Ticket #2167: Presenting project details on project summary page. Bug report. SourceForge. [Online]. Available: http://sourceforge.net/apps/trac/sourceforge/ticket/2167

[181] (2010, Jan.) Sourceforge collections. Website. FLOSSmole. [Online]. Available: http://flossmole.org/content/sourceforge-collections

[182] M. Squire. (2009, Dec.) [ossmole-discuss] SourceForge extract June 2009 – up to 10.000 projects missing. Mailing list. FLOSSmole. [Online]. Available: http://sourceforge.net/mailarchive/forum.php?thread_name=94d3bd120912052019i6d4894dcida05dd42c2803b9d%40mail.gmail.com&forum_name=ossmole-discuss

[183] M. Squire, K. Crowston, J. Howison, A. Wiggins, N. Radtke, and C. Mina. (2009, Jul.) [ossmole-discuss] SF collection strategies for the future (was: Re: Can't dump table). Mailing list. FLOSSmole. [Online]. Available: http://sourceforge.net/mailarchive/forum.php?thread_name=94d3bd120907050958h6274215718840a8943ffaf21f%40mail.gmail.com&forum_name=ossmole-discuss

[184] M. Squire. (2009, Oct.) [ossmole-discuss] project_statistics_60. Mailing list. FLOSSmole. [Online]. Available: http://sourceforge.net/mailarchive/message.php?msg_id=94d3bd120911221614x1fb8b91clf67ce8ceaaae064c%40mail.gmail.com

[185] (2011, Aug.) XFree86 Project. Website. [Online]. Available: http://sourceforge.net/projects/xfree86/

[186] (2011, Mar. 14,) XFree86 home to the X Window System. Website. The XFree86 Project, Inc. [Online]. Available: http://www.xfree86.org

[187] N. Ducheneaut, "Socialization in an open source software community: A socio-technical analysis," *Computer Supported Cooperative Work*, vol. 14, no. 4, pp. 323–368, 2005.

[188] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. San Jose, CA, USA: IEEE Computer Society, Oct. 11–14 2000, p. 131.

[189] A. Capiluppi, "Models for the evolution of OS projects," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Amsterdam, The Netherlands: IEEE Computer Society, Sep. 22–26, 2003, pp. 65–74.

[190] V. Grimm and S. F. Railsback, *Individual-Based Modeling and Ecology*. Princeton, NJ, USA: Princeton University Press, 2005.

[191] V. Grimm, K. Frank, F. Jeltsch, R. Brandl, J. Uchmanski, and C. Wissel, "Pattern-oriented modelling in population ecology," *Science of the Total Environment*, vol. 183, no. 1-2, pp. 151–166, 1996.

[192] S. F. Railsback and B. C. Harvey, "Analysis of habitat-selection rules using an individual-based model," *Ecology*, vol. 83, no. 7, pp. 1817–1830, Jul. 2002.

[193] T. Wiegand, F. Jeltsch, I. Hanski, and V. Grimm, "Using pattern-oriented modeling for revealing hidden information: A key for reconciling ecological theory and application," *Oikos*, vol. 100, no. 2, pp. 209–222, 2003.

[194] V. Grimm, E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H.-H. H. Thulke, J. Weiner, T. Wiegand, and D. L. DeAngelis, "Pattern-oriented modeling of agent-based complex systems: Lessons from ecology," *Science*, vol. 310, no. 5750, pp. 987–991, Nov. 2005.

[195] B. P. Zeigler, *Theory of Modeling and Simulation*. New York, NY, USA: John Wiley and Sons, 1976.

[196] W. M. Mooij and D. L. DeAngelis, "Uncertainty in spatially explicit animal dispersal models," *Ecological Applications*, vol. 13, no. 3, pp. 794–805, 2003.

[197] T. Wiegand, E. Revilla, and F. Knauer, "Dealing with uncertainty in spatially explicit population models," *Biodiversity and Conservation*, vol. 13, no. 1, pp. 53–78, 2004.

[198] R. Kicinger, T. Arciszewski, and K. A. De Jong, "Evolutionary computation and structural design: A survey of the state of the art," *Computers and*

Structures, vol. 83, no. 23-24, pp. 1943–1978, 2005. [Online]. Available: http://www.kicinger.com/publications/pdf/KicingerCAS2005.pdf

[199] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "MASON: A multi-agent simulation environment," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 82, no. 7, pp. 517–527, 2005.

[200] S. F. Railsback, S. L. Lytinen, and S. K. Jackson, "Agent-based simulation platforms: Review and development recommendations," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 82, no. 9, pp. 609–623, 2006.

[201] NetLogo users manual: FAQ. [Online]. Available: http://ccl.northwestern.edu/netlogo/faq.html

[202] High performance computing systems. ASU Advanced Computing Center. [Online]. Available: http://hpc.asu.edu/about/systems

[203] OpenABM consortium. OpenABM Consortium. Accessed: Jun. 12, 2011. [Online]. Available: http://www.openabm.org/about

[204] N. P. Radtke, M. A. Janssen, and J. S. Collofello, "What makes Free/Libre Open Source Software (FLOSS) projects successful? An agent-based model of FLOSS projects," *International Journal of Open Source Software and Processes*, vol. 1, no. 2, pp. 1–13, Apr.–Jun. 2009.

[205] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[206] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, L. M. Le Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[207] R. L. Thorndike, "Who belong in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, Dec. 1953.

[208] (2011, May 27) Wikipedia: Size comparisons. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia:Size_comparisons

[209] (2011, Jun. 21) Wikipedia: Size of Wikipedia. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

[210] (2011, Jun. 21) Wikipedia: Modelling Wikipedia's growth. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia:Modelling_Wikipedia's_growth

[211] R. P. Gabriel, "The Commons as new economy and what this means for research," *Upgrade: The European Journal for the Informatics Professional*, vol. VIII, no. 6, pp. 18–21, Dec. 2007.

[212] G. von Krogh, S. Spaeth, and K. R. Lakhani, "Community, joining, and specialization in open source software innovation: A case study," *Research Policy*, vol. 32, no. 7, pp. 1217–1241, 2003.

[213] (2011, Jul.) Software search. SourceForge. [Online]. Available: http://sourceforge.net/softwaremap/?&fq[]

[214] T. Dybå, B. Kitchenham, and M. Jorgensen, "Evidence-based software engineering for practitioners," *IEEE Software*, vol. 22, no. 1, pp. 58–65, 2005.

[215] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, Aug. 2002.

[216] L. M. Pickard, B. A. Kitchenham, and P. W. Jones, "Combining empirical results in software engineering," *Information and Software Technology*, vol. 40, no. 14, pp. 811–821, 1998.

[217] M. Mayer, "Fifty percent of what Google launched in the second half of 2005 actually got built out of 20% time," Seminar, Google, Standford University, Jun. 30, 2006, accessed: Aug. 14, 2011. [Online]. Available: http://www.youtube.com/watch?v=soYKFWqVVzg

[218] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. (2001, Feb. 11–13,) Manifesto for agile software development. [Online]. Available: http://agilemanifesto.org

[219] F. Ortega, J. M. Gonzalez-Barahona, and G. Robles, "On the inequality of contributions to Wikipedia," in *HICSS '08: Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. Washington,

DC, USA: IEEE Computer Society, 2008, pp. 304–310. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2008.333

[220] J. Voss, "Measuring Wikipedia," in *Proceedings of the 10th International Conference of the International Society for Scientometrics and Informetrics*. ISSI, Jul. 2005, pp. 221–231.

[221] (2011, Jul. 14,) List of Wikipedias. Wikimedia Foundation, Inc. San Francisco, CA, USA. [Online]. Available: http://meta.wikimedia.org/wiki/List_of_Wikipedias

[222] (2011, Aug. 14) Wikipedia. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia

[223] D. Anthony, S. W. Smith, and T. Williamson. (2005, Nov.) Explaining quality in Internet collective goods: Zealots and good samaritans in the case of Wikipedia. [Online]. Available: http://web.mit.edu/iandeseminar/Papers/Fall2005/anthony.pdf

[224] A. Lih, "Wikipedia as participatory journalism: Reliable sources? Metrics for evaluating collaborative media as a news resource," in *Proceedings of the 5th International Symposium on Online Journalism*, Austin, TX, USA, 2004. [Online]. Available: http://staff.washington.edu/clifford/teaching/readingfiles/utaustin-2004-wikipedia-rc2.pdf

[225] J. Giles, "Internet encyclopaedias go head to head," *Nature*, vol. 438, no. 7070, pp. 900–901, Dec. 2005. [Online]. Available: http://dx.doi.org/10.1038/438900a

[226] T. Chesney, "An empirical examination of Wikipedia's credibility," *First Monday*, vol. 11, no. 11, 2006.

APPENDIX A

COPYRIGHTED MATERIAL REUSE PERMISSION

Sections of text in this dissertation appear in the International Journal of Open Source Software and Processes edited by Stefan Koch Copyright 2009, IGI Global, www.igi-pub.com. Posted by permission of the publisher.

Request from Author for Reuse of IGI Materials

IGI Global ("IGI") recognizes that some of its authors would benefit professionally from the ability to reuse a portion or all of some manuscripts that the author wrote and submitted to IGI for publication. Prior to the use of IGI copyrighted materials in any fashion contemplated by the IGI Fair Use Guidelines for Authors, the author must submit this form, completed in its entirety, and secure from IGI the written permission to use such materials. Further, as a condition of IGI providing its consent to the reuse of IGI materials, the author agrees to furnish such additional information or documentation that IGI, in its sole discretion, may reasonably request in order to evaluate the request for permission and extent of use of such materials.

IGI will consider the special request of any author who:

- Completes, signs and returns this form agreeing to the terms; and
- Agrees that any IGI copyrighted materials will be labeled with the standard IGI identification of: **"This chapter/paper appears in <fill in publication title here> edited/authored by <fill in name of author/editor here> Copyright 2007, IGI Global, www.igi-pub.com. Posted by permission of the publisher."**

Title of article/chapter you are requesting: What Makes Free/Libre Open Source Software (FLOSS) Projects Successful? An Agent-Based Model of FLOSS Projects

Title and book author/editor where this IGI material appears: International Journal of Open Source Software and Processes, ed. Stefan Koch

Purpose of request (where this material will appear):

- ☐ Posted on a secure university website for your students to access in support of a class. (Posted paper must carry the IGI Global copyright information as outlined above.)
- ☐ Posted in a university archive. The Website address is: http://
- ☐ Posted on a personal Website: The Website address is: http://
- ☐ Republished in a book of which I am the editor/author. Book title of proposed book:

  Publisher publishing proposed book:
- ☒ Other purpose (please explain):

  For inclusion in my Ph.D. dissertation

With your signature below, you agree to the terms stated in the IGI Global Fair Use Guidelines. This permission is granted only when IGI returns a copy of the signed form to you for your files.

Your name: Nicholas P. Radtke
Your signature: Nicholas P. Radtke
Organization: Arizona State University
Address: 1707 W. Heatherbrae Dr.
Phoenix, Arizona 85015
U.S.A.
Fax: N/A
E-mail: radtken@aztecfreenet.org

For IGI Use
Request accepted by IGI Global: Jan Travers
Digitally signed by Jan Travers
DN: cn=Jan Travers, o=IGI Global, ou=Vice President, email=jtravers@igi-global.com, c=US
Date: 2009.12.23 11:00:21 -05'00'
Date:

Please complete and mail or fax this request form to:
Jan Travers • IGI Global, 701 E Chocolate Avenue • Hershey PA 17033 • Fax: 717/533-8661