Clustering and Dependencies in Free/Open Source Software Development: Methodology and Tools[1]

Rishab Aiyer Ghosh (*rishab@dxm.org)*

Senior Researcher, University of Maastricht / MERIT

(www.infonomics.nl)

Abstract: *This paper addresses the problem of measurement of non-monetary economic activity, specifically in the area of free/open source software[2] communities. It describes the problems associated with research on these communities in the absence of measurable monetary transactions, and suggests possible alternatives.. A class of techniques using software source code as factual documentation of economic activity is described and a methodology for the extraction, interpretation and analysis of empirical data from software source code is detailed, with the outline of algorithms for identifying collaborative authorship and determining the identity of coherent economic actors in developer communities. Finally, conclusions are drawn from the application of these techniques to a base of software.*

[1] An early version of this paper was presented at the IDEI/CEPR Workshop on "Open Source Software: Economics, Law and Policy", June 20-21, Toulouse, France. Support for this revised version was drawn from the Project on the Economic Organization and Viability of Open Source Software, which is funded under National Science Foundation Grant NSF IIS-0112962 to the Stanford Institute for Economic Policy Research. (see http://siepr.stanford.edu/programs/OpenSoftware_David/OS_Project_Funded_Announcmt.htm)

[2] "Free Software" is the original, correct and more popular term by far among practitioners (http://floss1.infonomics.nl/stats.php?id=1) but is functionally equivalent to the more recent, publicity and business-friendly term "open source". Given the apparent political weight given to these terms (http://floss1.infonomics.nl/stats.php?id=2) we use the neutral abbreviation FOSS, and unless specified, mean both terms wherever any one is used.

Table of contents

1. Defining the problem: non-monetary implicit transactions ... 3
1.1. Background: the best things in life are free? ... 3
1.2. When a gift is not a gift ... 3
2. Forms of research and the role of measurement ... 4
2.1. Free software as a way of life ... 5
2.2. Open Source as a way of work ... 5
2.3. Open Source as a way of software development ... 6
3. Where is the data on FLOSS activity? ... 7
4. Non-monetary measurement ... 8
5. Free software developers: a starting point for measurement ... 10
5.1. How software tells its own story ... 10
5.2. What may be inferred ... 11
6. What is in the source: extracting empirical data from software source code ... 12
6.1. Authorship information ... 13
6.1.1. Alternative methods ... 14
6.1.2. Project “culture” and code accreditation ... 14
6.2. Size and integrity ... 15
6.3. Code dependency between packages ... 16
6.3.1. Identifying function definitions as an aid to dependency analysis ... 17
6.4. Clusters of authorship ... 18
6.4.1. Building clusters ... 20
6.4.2. Analysing clusters: collaborators and non-collaborators ... 21
6.4.3. Clustering and dependency: cause and effect? ... 22
6.5. Technical details: summary methodology and data structure ... 23
6.5.1. Table 2: Stages of analysis and resulting data format ... 23
7. Conclusion, Outlook, and practical considerations ... 24
7.1. The FLOSS source code scan / Orbiten 2 ... 24
7.2. LICKS: Studying multiple versions of the Linux kernel ... 25
8. Annexure: References and Literature ... 27

1. Defining the problem: non-monetary implicit transactions

1.1. Background: the best things in life are free?

One notable aspect of open source software development is the hidden – and often non-existent – role played by money. Although not unique in the context of the Internet, this does complicate an understanding of open source as an economic phenomenon. Subjecting this phenomenon to an economist's analysis does not necessarily require the abundant and overt use of money, but it does require the ascribing of some economic motives to participants. It also requires methods of measurement, to substitute for the measurement otherwise provided by a visible use of money.

Furthermore, most descriptions of the open source phenomenon rely on ideology or anecdotes, rather than any hard data. Non-monetary methods of economic measurement are a way of providing such data, and current work explores and implements such methods in the open source context (admittedly with the aim, in part, of backing up anecdotal models of the phenomenon's functioning). Some measurement indicators include: concentration of contribution and authorship; formation of collaborative author communities; dependency and "trade" between such communities[3].

1.2. When a gift is not a gift

The precise degree and influence of altruism or gift-giving – which are quite separate things – on open source is a matter for further research. What is clear is that the analysis of the open source phenomenon is complicated by the fact that it is not a priced market, nor is it well described by the literature on barter exchanges. Literature in that field is rare. A model describing the free software/open source phenomenon as a "cooking-pot market" of largely non-monetary economic activity explains why the transactions are implicit[4].

[3] Orbiten Free Software Survey OFSS01, May 1999; OFSS02, February 2002, http://orbiten.org; Free/Libre and Open Source Software Study (EU project FLOSS), on-going, www.infonomics.nl/FLOSS/

[4] This model is based on two observations: first, though of obvious utility to consumers, information products on the Internet have near-zero marginal costs of duplication and distribution for their producers though there may be significant one-time costs of creation. Second, the universe of "free" collaborative content production on the Internet may lack visible one-to-one transactions, but the continuing awareness of a quid pro quo among participants suggests that transactions are implicit rather than absent. The "cooking-pot" model hypothesises that participants contribute their products to a delineated commons, or "cooking-pot", in a sort of exchange – with implicit one-to-many transactions – of the one-time production cost with the value gained from individual access to a diversity of products contributed by others. There are other parallel motives for contribution, but this is one of the main *economic* ones, and also happens to be in some sense quantifiable. This model is described in detail in Ghosh 1998.

A comparison of some of the properties of priced markets, barter exchanges and cooking-pot markets is listed in table 1. It shows that if open source follows the cooking-pot market model, research into its functioning is significantly hampered by the lack of quantifiable data points, the implicit nature of transactions, indirect nature of rewards, and above all the inability to use existing, well-tested tools and techniques for measurement, analysis and modelling. This handicap runs through all the lines of research further described below.

Table 1: What sort of market? A comparison

Priced markets	Barter exchanges	Cooking-pot markets
Price tags	No price tags, but exchange rates for types of barter goods – i.e. relative price tags – are available	No price tags
Priced transactions – buying/selling – at every step	Identifiable barter transactions – trades – at every step	Few identifiable transactions – production is gratis, and value is received from the whole network with no explicit linkage; only a *value flow*[5]
Quantifiable by price and volume (number of transactions)	Partly quantifiable by relative size and number of transactions	Not directly quantifiable – no price tags to add up; no transactions to count
Tangible benefits direct by proxy (money to buy other things); quantifiable	Tangible benefits direct (end-products exchanged); partly quantifiable	Tangible benefits indirect and hard to quantify
Intangible benefits less important, less direct?	Intangible benefits less important, less direct?	Intangible benefits apparent (reputation; sense of satisfaction, etc) but indirect

2. Forms of research and the role of measurement

Human activity can be studied from various perspectives: collective action can be explained as the functioning of a rational economic marketplace, a law-abiding jurisdiction, or a community with a common belief-system, for example. Similarly, individual action can be credited to rational economic self-interest; subscription to a set of rules or fear of punishment for their violation; or an altruistic satisfaction in the good of others[6]. Often, the same act can be described from an economic, legal or sociological perspective ascribing motives to the acting individual or collective that may all coexist.

[5] Ghosh, 2003

[6] Ghosh 1996

Which motives to credit – and hence which method of study to use – is determined by the purpose of study, the degree to which the actors are conscious of their motives and of course the interest of the persons conducting such study.

Research into free/open source software depends on these different perspectives towards defining what free/open source is. Since it is many things to many people, often all at the same time, various aspects of open source are researched into in various ways. I classify these as follows: Open source as a way of life; a way of work; and a way of software development – but all are dependent on measurement of dynamics within the developer communities.

2.1. Free software as a way of life

FLOSS is often termed, by its own adherents, as a philosophy (as in: "the Free Software Philosophy"[7]). Open source participants adopt a rationale for their actions different from that of their peers in the proprietary software world. (Seemingly, at any rate; there are of course many who are active in proprietary as well as open source worlds.) The concepts of *community, gift, shared ideals* and so forth are often brought up by FLOSS participants as well as researchers into the phenomenon. The existence, validity, role and degree of influence of such concepts are topics that should be further investigated, with a return to first principles of social anthropology. The motives that drive open source participants clearly determine the development of the phenomenon as an economic or socio-legal force, and thus have much bearing on the following two lines of research.

Motives play a role, of course, in any monetary economic scenario as well – however, there not only is profit maximisation as the leading motive assumed, it can also be backed up and explained with the existence of empirical data (such the presence of profit margins in any market). Such monetary data is not available for FLOSS communities except at the anecdotal level for a very unrepresentative sample subset ("star programmers" and the rare profitable open source company). The main reason for this lack of data is the fact that monetary transactions are largely non-existent within the FLOSS production process. Given this absence, any other form of data that can provide an empirical basis for arguments on motivation, sense of community, social and political coherence is clearly crucial.

2.2. Open Source as a way of work

It is a fact. There are people who make a living *not* being paid cash for software they write. This raises many questions. One set is related to survival in the monetary economy

[7] http://www.gnu.org/philosophy/philosophy.html

("How do they order pizza?")[8] – what sort of a living do such people make; who pays them and why; what benefits accrue to employers of such people who pay for what may be freely available. Another set of questions is related to the non-monetary economy that results from the production of goods without payment – if they're not receiving cash for their software, what, if anything, do they receive instead? And how much? What do they give for access to other free software?

The value of money as a measuring tool is immeasurable. Lacking this, alternatives must be found in order to identify power structures; ownership and effective control of systems; vulnerabilities and dependencies in the "economy" surrounding open source systems. To illustrate: Microsoft's position in the economy is easy enough to analyse, since its property and influence is quantified in monetary terms; but the position of Linus Torvalds or the Apache team is hard to quantify even within the open source community (let alone in the economy at large), even if a system of definable reputation is accepted as a way of doing so.

The further development of measurement and modelling methods is, therefore, crucial to the understanding and better functioning of open source, and its integration into the monetary economic system. Examples of such measurement methods were given at the beginning of this paper and are elaborated in the section on methodology and techniques, but obviously the development of new methods is an area for further research.

2.3. Open Source as a way of software development

Finally, open source is a method of developing software. It is often quite different from the formal development methodologies of proprietary software companies. The element of collaborative authorship is much discussed; less so is the element of competition (proprietary software, developed in-house in a single firm, is directed from above and lacks the non-coercive Darwinism of bottom-up distributed software development).

As a software development technique open source faces several challenges: intellectual property rights (use of proprietary IPR as well as the protection of open source IPR); software quality and reliability; version control and responsiveness to environmental changes; credit and liability management.

It is as a development technique that open source has been most effectively studied. Indeed, the best-known open source literature is far more useful as a study of open source as a

[8] See "Can you eat goodwill?" in Ghosh, 1998; also, interview with Linus Torvalds 1996 & 1998, published in *First Monday,* www.firstmonday.org/issues/issue3_3/

software development model than of economics or anything else.[9] This is only to be expected, as most people whose expertise lies closest to open source software development (as participants themselves) do not have a significant expertise in socio-economics, but they can share their development experiences, while most socio-economic researchers are not only very new to the field of open source, but face a complete lack of empirical data resulting in a literature largely based on anecdotes.

3. Where is the data on FLOSS activity?

The absence of empirical, factual and verifiable data on a large scale is clearly a major disadvantage to most kinds of research into the FLOSS phenomenon. Previous experience from the very few extensive surveys carried out so far[10] suggest that quantitative and qualitative empirical survey methods can be useful. However, surveys can introduce biases that can be difficult to calculate. Tools and systems to analyse the traces left behind by collaborative activity in the form of software source code or discussion archives – Internet archaeology? – can be much more useful in finding *hard facts* (Ghosh & Ved Prakash, 2000). But such an “Internet Archaeology” is just beginning.

The free software (or open source) "community" is much talked about, though little data on this community and its activities is available. Free software and open source are considered competing definitions of this community or phenomenon. For researchers into this phenomenon, especially economists, the fact that software is free[11] is what makes analysis difficult since tools for measurement without the use of money are not sufficiently advanced. However, the fact that software source code is open provides a solution. Source code is a fact: it includes pure hard data that can be gathered through automated analysis producing results far more objective than any sample-based interactive survey.

The Orbiten Free Software Survey[12] in May 2000 first developed the basic software tools and methodology to conduct an automated scan of software source code to extract a body of empirical data suitable for analysis and description of the free software/open source community.

These software tools have been further developed for a second Orbiten survey, as part of the FLOSS project[13] funded by the European Commission’s IST programme[14], and are described further later in this paper.

[9] see e.g. Raymond 1998

[10] Robles 2001, FLOSS 2002, BCG 2002

[11] Used here in the sense of “without payment” – *not* in the sense of “freedom”, which is the sense intended by the originators of the term Free Software

[12] http://www.firstmonday.org/issues/issue5_7/ghosh/index.html

[13] http://www.infonomics.nl/FLOSS/

[14] http://www.cordis.lu/ist/

4. Non-monetary measurement

To measure price-based economic activity is straightforward – individual transactions are clearly identified, transactions can be tracked and collated into defined and segmented markets, and the activity on price-based markets can be measured. By definition, there is always a price on everything.

Without money as a tool of measurement, you must find other ways of quantifying value, and you must identify the different systems of ascribing value and exchange rates between them. Furthermore, without identifiable explicit transactions you don’t have explicitly identifiable transactors – i.e. you don’t know who’s doing the valuing, the production, or the consumption.

The broad question of identifying *who* is doing *how much* of *what* with *whom* is something that gets concisely focused in a price market, by examining a transaction, a price, or a collection of transactions in a market.

On a cooking-pot network such as the FLOSS developer community, without explicit identifiable transactions, the “**who**” is a non-trivial question, as there are no clearly identified transacting parties to any identifiable exchange. The closest equivalents to transacting parties are contributor groups, such as “the Linux developer community”.

Such amorphously defined groups change shape, and often have a *radiating identity* – with a central core of group members where contribution peaks, surrounded by members of reducing contribution and group identity. The sharpness of the contrast between centre and surround, peak and valley, is what defines the cohesiveness of the group, and determines its suitability for treatment as a single economic entity. Identifying these groups involves *measuring* contribution levels and *clustering* contributors.

Understanding the interaction *between* economic entities – who is doing **what** with whom – involves studying shifts in patterns of membership and cross-membership between different groups. In the case of FLOSS, for instance, the economic dependence and value flow across free software / open source projects such as Linux, Perl and various parts of GNU/FSF can be mapped across time by tracking the authorship of program source code and identifying author group membership. Tracking authorship is quite easy in theory, though it isn’t very easy in practice since authors aren’t always easy to identify, as the discussion in section 6.1, Authorship information, shows. Once authors are identified their group membership can be determined by following them and their source code components across projects.

There is also the possibility of identifying dependencies directly from activity traces in Internet communities, or in software source code, in a form of “Internet archaeology”. Internet communities have explicit or implicit references encoded into individual postings, and mapping them is a non-trivial but feasible task[15]. So does software source code, where individual components refer to others in a fairly explicit process. Indeed, dependency as shown in function calls was the subject of a well-known poster available on thinkgeek.com, based on an analysis of the Linux kernel by Paul "Rusty" Russell[16]. At a somewhat less detailed level, it is also possible with appropriate heuristics to identify the dependency between source code packages – and to impute from that the dependency and "trade" between clusters of authors contributing to such packages.

Tracking such *value flow* could make it possible to model and predict group membership, flux in exchange and membership between groups. It could even help identify the value *exchange rate* across groups – there is a measurable value given to GNU/FSF software within Linux groups, which translates to value placed on authors of such software (a proxy for which is "reputation"); but probably much less value is given by Linux programmers to members of and content from, say, rec.music.early, a newsgroup on early music.

Measures of contribution, its concentration and distribution within groups can help model shape changes within groups – enabling one to predict their disintegration, for instance, or pressures towards commercialisation or guild-type segmentation (by shutting out non-members).

Determining who is doing **how much** is partly a problem of quantification. Although no simple measure such as number of transactions or price levels is possible, other indicators of value flow and proxies for value can be used.

For example, it is possible to monitor producer/consumer activity and concentration by area – such as population, frequency, distribution and overlap among participants in Linux and perl developer communities, or readers of newsgroups such as rec.pets.cats and rec.music.classical.

More practically useful measures are possible. The health of a cooking-pot economy can be measured through "macroeconomic" means. These could include: the *lurker coefficient*, indicating the concentration of active participants within a group and arrived at by calculating the ratios of contributions to contributors. This relates to what may be called free riding, but for open source, or Internet communities in general, free riding may be the wrong

[15] Marc Smith, Microsoft Research, http://netscan.research.Microsoft.com
[16] E-mails on file with author. Poster available at http://www.thinkgeek.com/stuff/fun-stuff/3884.shtml

term since it implies a one-way transfer of value (to the free-riders) while *lurkers* are often thought to bring value to a community.[17]

However, a high lurker coefficient may affect the motivation of the relatively small number of active participants to contribute free of charge and hence encourage barriers, analogous to the formation of guilds – or a shift to price-based model, as in the case of the Internet Movies Database, which was free and entirely non-monetary when users were active contributors, but is now advertising-based[18].

Equivalence measures, quantifying links between information exchanges and price-based markets outside, are possible too. These could be based on time spent in "free" production or by comparing equivalent priced products, where applicable.

The non-monetary-with-implicit-transaction characteristics of cooking-pot markets are ever present on the Internet. Where to start trying out new forms of measurement of such economic activity? Free software seems an obvious choice.

5. Free software developers: a starting point for measurement

In the past few years there have been some surveys conducted of developers, though usually on fairly small samples and far from comprehensive. No survey actually looks at what is perhaps the best source of information on free software (and the only source of objective information) – the source code itself. This was attempted first as an experiment in late 1998 developed into the Orbiten Free Software Survey[19]. Although there have since been other surveys of authorship[20] and many of the relatively recent web sites that provide an environment for open source development such as SourceForge[21] provide some statistics, these often do not adopt the approach of looking at the free software community from the bottom up – from the facts as they are created, rather than as they are reported.

5.1. How software tells its own story

The Orbiten Survey took advantage of one of the key features of the software development community. In contrast to other non-monetary exchange systems ("cooking pot networks") on the Internet such as newsgroups and discussion forums, much of the activity around is precisely recorded. The "product" – software – is by nature archived. Since source code is available, the product is open to scrutiny not just by developers, but also by

[17] E.g. Linus Torvalds on how the (non-paying, "free-riding") user base for Linux is "actually a larger bonus than the developer base" quoted in Ghosh 1998.

[18] "Is reputation a convertible currency?" in Ghosh 1998

[19] Ghosh, 1998

[19] Ghosh & Ved Prakash, 2000

[20] Dempsey et al. 2002, Also, FLOSS 2002 Part V was a follow up to the 2000 Orbiten source code survey.

economists. Arguably all economic activity: production, consumption and trade – in the Internet's cooking-pot markets is all clearly documented, as it is by nature in a medium where everything can be stored in archives.

The difference between software and discussion groups – where too the "product", online discussions, is available in archives – is that software is *structured*. To understand what is going on in a discussion group, one might need to read the discussions, which is quite complicated to do in an automated fashion. However, reading and understanding software source code is by definition something that is very easily done by a software application.

Software source code consists of at least three aspects that are useful for economic study. It contains *documentation* – the least structured of all the data here, since it is written in a natural language such as (usually) English. This provides information on among other things the authorship of the software. *Headers* are called different things in different programming languages but perform the same function, of stating dependencies between the software package under scrutiny and other software packages. Finally, the *code* itself provides information on the function of the software package. As an automated interpretation of this is exactly what happens when the program is compiled or run, there may be far too much information there to be usefully interpreted for an economist's purpose. But it is possible to have an idea of the importance or application domain of the code in some subjective (if well-defined) sense – it works with the network, say, or has something to do with displaying images.

Naturally these categories are not sharply divided – indeed most authorship information for individual components of a software package may be present through comments in the code, which fits, for current purposes, the category of documentation.

There are formalized procedures for authors to declare authorship for entire packages on certain repositories and archives, but such information needs to be treated carefully too[i]. The data may be reliably present, but its semantics are variable. Usually such "lead authors" hold responsibility for coordination, maintenance and relations with a given repository, but data on other collaborating authors – let alone authorship of individual components – may be entirely missing. On the other hand such detailed data are usually present in the source code itself.

5.2. What may be inferred

There is little point doing a small "representative" survey since results are meaningless unless large amounts of software are processed. Given the data at hand, and the

[21] www.sourceforge.net

degree of structural complexity for automation, there is a cornucopia of interesting findings to be made. At the very simplest, a map of author contribution can be made, resulting in an indicator of the distribution of non-monetary "wealth", or at any rate production. This is in theory simple to do – count the lines of code and attribute that figure to the author(s) with the nearest claim of credit.

More complicated is to look for links between projects and groups of projects, as well as links between groups of authors. The former can be done by looking for dependencies in the source code – references from each software package to other software packages. The latter is inferred through the identification of authors who work on the same project or group of projects. Of course both these indicators refer to one another – projects with related authors are in some way related projects; authors of a project that depends on another project are in a way dependent on that other project's authors.

Measuring such dependencies and interrelationships can provide an insight into the tremendous and constant trade that goes on in the free software cooking-pot network, and can probably also provide an indicator of the relationship with commercial software and the (monetary) economy at large. Finally, the value of all such parameters can be applied over the fourth dimension, either using a simple chronology of time, or the virtual chronology of multiple versions of software packages, each of which replaces and replenishes itself wholly or in part as often as every few weeks.

6. What is in the source: extracting empirical data from software source code

We proceed to look further into the details and format of empirical data that can be extracted through a primarily automated scan of software source code. The degree (and reliability) of extractability, as it were, depends on the type of data extracted. These fall into four broad categories.

- Authorship information for source at the sub-package/component level
- Size and integrity information for source code at the package level[22]
- The degree of code dependency between packages
- Clusters of authorship: groups of authors who work together, identified by their joint work on individual packages

[22] WIDI 2000; Jones 2002

[22] a package, loosely defined, is several files distributed together. Usually a package can be reliably dated to a specific version or release date. Sub-packages are the individual files or collections of files at the next lower level(s) of the distribution directory structure

All these data can also be collected chronologically, i.e. over different versions of source code or of source packages at different points in time.

6.1. Authorship information

Authorship information is perhaps the most interesting yet least reliable of the data categories. Although most FOSS developers consider marking source code they've written as important[23] they apparently do not take sufficient care to do so in a consistent manner. Claiming credit is usually done in an unstructured form, in natural-language comments within source code (such as "written by", "author" or copyright declarations), posing all the problems of automated analysis of documentation. Several heuristics have been used, however, to minimise inaccuracies and are described further in the technical documentation for the software scanning application CODD[24].

Particular issues or biases that have not yet been fully resolved include several cases of "uncredited" source code[25]. This is either a result of carelessness on the part of authors, or in some cases, a matter of policy. Developers of the web server Apache[26], for instance, do not sign their names individually in source code. A large amount of important source code is the copyright of the Free Software Foundation, with no individual authorship data available[27]. Although one must be careful to tailor credit extraction methods to specific source code packages if highly detailed analysis is to be performed, the integrity of the data in general is not necessarily affected by the method described above. Indeed, in general this method of determining authorship by examining the source code itself shares (some of) the bias of alternative methods towards crediting lead authors, as many authors who contribute small changes here and there do not claim credit at all, handing the credit by default to lead authors[28].

[23] According to the FLOSS developer survey, 57.8% consider it "very important" and a further 35.8% don't consider it "very important" but claim to mark their code with their names anyway; see http://floss1.infonomics.nl/stats.php?id=31

[24] Designed by Rishab Ghosh and Vipul Ved Prakash. Originally implemented by Vipul Ved Prakash; further developed and currently maintained by Rishab Ghosh and Gregorio Robles. The first version of CODD was created in 1998 and and stands for"Concentration of Developer Distribution". See also http://orbiten.org/codd/

[25] This is a significant, but not a huge fraction of code: in the scan of over 22,000 projects for the FLOSS survey, about 10% of code was uncredited; for a scan of 3 versions of the Linux kernel, about 14% was uncredited. In the original Orbiten survey of 3,149 projects, 8% was found uncredited. See FLOSS 2002 Part V, Ghosh & David 2003, Ghosh & Ved Prakash 2000.

[26] www.apache.org

[27] Several authors formally assigned their copyright to the FSF in order to protect themselves from liability and increase the enforceability of copyright. Assignment records are not yet available for access to academic research.

[28] There is a counteracting bias introduced by the CODD heuristics, which usually give equal credit to multiple authors when they are listed together with no identifiable ranking information (thus narrowing the difference between a lead author and a minor author in case they are listed jointly).

6.1.1. Alternative methods

There are alternative methods of assessing authorship of free/open source software. Typically, they are based on more formal methods of claiming credit. In the Linux Software Map, for example, it is usually a single developer who assumes the responsibility for an entire package or collection of packages that are submitted to an archive. On collaborative development platforms such as SourceForge, similar methods are used; specific authors start projects and maintain responsibility for them. With these methods, assessing authorship is limited to collating a list of "responsible" authors. Clearly the semantics of authorship here are quite different from what we have previously described, since "responsible" authors may be responsible for maintenance without actually authoring anything, and in any case there are several contributors who are left out of the formal lists altogether. Thus, any attempt at identifying clusters of authors is likely to fail or suffer considerable bias.

A more detailed and less biased (but also less formal) method of author attribution is used by developers themselves during the development process. Either through a version-control system, such as CVS or Bitkeeper[29], or simply through a plain-text "ChangeLog" file, changes are recorded between progressive versions of a software application. Each change is noted, usually with some identification of the person making the change – in the case of a version control system this identification, together with the date, time and size of change is more or less automatically recorded. However, again the semantics vary – most projects limit to a small number the people who can actually "commit" changes, and it is their names that are recorded, while the names of the actual authors of such changes may or may not be.

Naturally, no method is perfect, but the purpose of the above summary is to show that formal author identification methods do not necessarily provide much additional clarity into the nature of collaborative authorship, while introducing their own biases. Depending on the specific analysis planned and the level of detail, an appropriate credit data extraction system must be chosen. In general, for a varied and large spectrum of source code, the CODD-Ownergrep method seems to be the most accurate[30].

6.1.2. Project "culture" and code accreditation

It is important to note that applying any of these tools to free/open source projects requires an understanding of the "culture" of each project. The more detailed the data and the more one intends to interpret it, the better such understanding needs to be, since source code, versioning information and so forth are not perfectly formalised. Coding styles and

[29] CVS: Concurrent Versions System, http://www.cvshome.org; Bitkeeper: http://www.bitkeeper.com

[30] Furthermore, CODD's credit extraction system is being extended to incorporate CVS and Bitkeeper version tracking information, which would provide an alternative reference point for credit data.

organisational conventions differ from project to project, with the result that the same data can have different semantics in different projects. This is one reason the CODD-Ownergrep method is likely to remain useful for a long time – although it doesn't always provide very much detail, it is arguably more comparable across a range of projects and versions than any other method.

Using versioning information as described above provides much more depth of information – including authorship credits on a per-line basis – but also provides more data points that may be subject to incorrect or over-interpretation without a thorough understanding of conventions in each project's "culture". This is especially true for versioning data since versioning systems don't necessarily distinguish between "authors" and "committers" (the "editors" who actually approve and enter a submitted change into the source code)[31]. As a result, versioning data can be less comparable across projects and more suited to in-depth study of specific, carefully chosen project cases. Without adjusting for project "culture", it is easy to over-interpret data and come to incorrect conclusions, as described by Tuomi[32].

6.2. Size and integrity

There are many ways to value the degree of production a specific software package represents. Especially when it does not have a price set on it, the method of choosing an attribute of value can be complex. One value, which makes up in its completely precise, factual nature what it may lack in interpretability is size. The size of source code, measured simply in bytes or number of lines, is the only absolute measure possible in the current state of F/OSS organisation and distribution. Specifically, measuring the size of a package, and the size of individual contributions, allows something to be said about the relative contributions of individual authors to a package, and of the package to the entire source code base. It may also be possible to impute time spent in development or some a monetary value based on size.

Size is not the only (or always the most accurate) measure of the value of a given body of source code. Functionality is another useful measure, and the software engineering literature abounds in attempts to develop indicative measures of functionality, notably *function-point* analysis[33]. A less versatile but far simpler method that scales well and can be applied to source code is to count function definitions. A function (also called a procedure or method) is the smallest reusable unit of code consistent across several programming

[31] A comparison of CVS and CODD-Ownergrep on certain projects highlight the distinction between author and committer and indicate how CODD-Ownergrep may be a more generally reliable method of determining authorship – see Robles et al 2003

[32] Tuomi 2002

languages. Although determining where a function is defined in the source code is certainly language-specific, the fact is that most F/OSS code is in C or C++, and therefore can be scanned for function definitions according to the syntax of those languages (similar scans can be performed for other common – though less popular – languages such as Perl or Python). Studying author contribution to source code through function counts rather than bytes of code provides at worst an alternative perspective, and at best a new set of useful indicators of how contribution can be valued.

In order to calculate the size of a package it is important to try to ensure its integrity. A given package – especially on development platforms – usually includes derivative or "borrowed" works that have been written separately by other developers, but may be required in order to for the package to run. These are not necessarily identified as "borrowed" and could, in theory, be counted twice. Furthermore, they can artificially inflate the apparent contribution of an author of a "borrowed" work. CODD tries to resolve this by identifying duplicate components across the entire scanned code base and allocating them to only a single package wherever possible. This promotes integrity and avoids double-counting, and also provides information useful for finding dependencies between packages, by replacing "borrowed" works with external references to those works.

6.3. Code dependency between packages

Since software is by nature collaborative in functioning, software packages usually depend on features and components from several other packages. Such dependencies must be explicitly detailed in a way that they can be determined automatically, in order for an application to run. As such, these dependencies can be identified through automatic scanning; indeed there are several developers' tools that serve this purpose. Such tools normally provide a high level of detail regarding dependencies (i.e. at a function call level), which may not be required for the purpose of analysis. Author credit information is rarely available at anything more detailed than file level, so dependency information at a more detailed level may not necessarily be very useful. Moreover, such detailed analysis would be computationally exceptionally hard to perform for 30,000 software packages!

Dependency analysis, like the other CODD tools, can be applied at varying levels of granularity. Due to the rather flexible definition – in developer terminology, as well as in this paper – of a "package", the method of encoding, and thus also of determining dependency links can differ widely across code samples. While sampling the Linux kernel code base, "packages" are components of the Linux kernel and tightly integrated; when sampling a

[33] Longstreet 2001. For the use of function points in the estimation of value production in national accounts, see Grimm et al 2002.

Linux software distribution such as Red Hat or Debian, the entire Linux kernel itself may be treated as a single "package", with much looser links to other packages within the code base. When code is looked at in more detail and packages are tightly integrated, the method for identifying dependencies is very different from when packages are examined at a more abstract level.

It is possible, however, to develop simple heuristics to identify dependencies at the package level[34]. One method is to retain information on duplicate files and interpret that as dependency information: if package *P* contains a file that has been "borrowed" from package *Q* where it originally belongs, *P* is dependent on *Q*.

Another method is based on header files. As described earlier, headers (called different things in different programming languages) define interfaces to functions, the implementations of which are themselves embodied in code files[35]. In order to access externally defined functions, a code file must *include*[36] a declaration for it, typically in the form of a statement referring to a header file. This is treated by CODD as an external reference. Various heuristics are used to identify the package where header file functions are actually implemented, and external references are resolved as links from one package (which *includes* or the header file and calls the functions declared in it) to another package (which defines the functions declared by the header file).

6.3.1. Identifying function definitions as an aid to dependency analysis

One accurate, but time-consuming heuristics that can be used for this task is to identify and map function definitions (see above, Size and integrity, p. 15). This way, a database is created with information on all identified functions defined in the source code, keeping track for each function the code file and package in which it is defined, as well as the header file in which it is declared. When CODD finds a dependent code file that includes a header file, it matches the functions declared in that header file (and potentially used by the dependent code file) to the various supporting code files that define those functions. Obviously, this technique is suitable only for relatively small projects as the resources consumed by this process grows exponentially for larger projects[37].

[34] There are developer tools which do this, producing different outputs for different purposes. This section illustrates a simple way of performing such dependency analysis as implemented in CODD.

[35] For the C/C++ programming languages, which amount for the largest proportion of general-purpose F/OSS, files ending with ".h" or ".hpp" are headers and those with ".c" or ".cpp" contain implementation code, while scripting languages such as Perl or Python do not use separate header files.

[36] Using the *#include* command in C/C++ source code, and other methods in other programming languages, such as *use* in Perl.

[37] For the LICKS project, when applied to Linux kernel version 2.5.25, this method identified over 5 million function dependencies for some 48,000 functions defined across more than 12,000 source code files. This was then summarised to 8,328 dependencies between 178 projects. Although the Linux source code was only about 175 Mb, over 600Mb of dependency data was generated. See Ghosh & David 2003

The above description should illustrate that it is possible to build dependency graphs for source code; although there may be several methods of doing so[38], the resulting dependency information is in any case very useful. Arguably a small package that is required by several others is more valuable (or valuable in a different way) than a large package without dependents. So further analysis of dependency information is very useful in order better to gauge the value distribution of packages – especially if this can be combined with information on authorship.

6.4. Clusters of authorship

Collaborative authorship implies that authors collaborate, i.e. that they author things together[39]. The degree of such collaboration is yet another attribute that can be found in the source code. Indeed, it is found through the analysis of data extracted from the first two steps described above, author credits and contribution size. The purpose of identifying clusters of authorship is simple: from individual actors with unclear motives and indeterminate flows of value to other actors, authors are transformed into somewhat more coherent groups whose interaction and inter-dependence becomes much easier to measure. Moreover, with chronological analysis the movement of individuals within and between groups can also be mapped, providing an insight into the functioning and behaviour of the entire F/OSS development system.

As described in a previous section (see "*radiating identit*y", page 8) amorphous groups of collaborators tend to cluster around a concentrated centre-point. In order to have maximum flexibility, as well as for practical reasons,[40] the identification of authorship clusters is carried out by a specially designed CODD-Cluster application[41].

[38] The RPM package system used by Red Hat Linux includes package-level dependency information that can be extracted easily; dependency information is also provided in similar packaging systems from other Linux distributors. At the other end of the spectrum, the utilities Cflow and Calltree provide detailed function-call-based dependency analysis – see http://barba.dat.escet.urjc.es/index.php?menu=Tools&Tools=Other

[39] What is found in the source code is not, strictly speaking, evidence of collaboration among authors, but their "co-participation" in the authorship of a given project or module – i.e. appearance of authorship credits for multiple authors of a single source code module. Other data sources, such as discussions on mailing-lists related to specific projects – can be used to prove collaboration among authors. However, there is a strong argument that "co-participation" in itself implies a high degree of collaboration. Collaboration is not necessarily implied in the case of joint authorship credit appearing in, say, academic papers, where it is possible (and common) for some of the co-authors to have only made comments, or written sections independently of other authors. However, for a computer program at the level of a single file or source code module, collaboration is a pre-requisite in order for the program to function at all, and any released version that would be available for analysis has necessarily gone through a process of coordinated modification where contributing authors have, in addition to control over their own contribution, some degree of awareness of (and control over, or at least acquiescence towards) the functioning of the rest of the module. Without such a degree of coordination there would probably not be a common released version, and the program would not function. This seems to justify the assertion that "co-participation" of authors in a given module, project or group of projects implies their collaboration.

[40] The computational difficulty of using standard statistical packages on 30,000 x 20,000 matrices in the case of identifying clusters across large code bases

[41] Designed by R. A. Ghosh and implemented by Ghosh, and Gregorio Robles.

The aim of this stage of data extraction – analysis, really, since no further raw data are extracted – is to identify clusters of authorship based on authors' degree of collaboration on common projects. This results in clusters of authors who work together[42]. It also results in equivalent clusters of projects – the result of the collaboration of authors in a given cluster.

Clustering is performed as a three-stage process. First, tables of project-wise author contribution are combined into a graph. This is a weighted bi-directional graph with projects (packages) forming the vertices. A link between two given vertices exists if there is at least one author common to the two projects. The weight of this link is calculated as a function of the degree of commonality (i.e. the number of common authors as a proportion of the total number of authors for both projects) as well as the degree of common contribution (projects whose common authors contribute more to each project are linked with a higher weight than projects with only minor authors in common). The link also preserves information regarding the common authors themselves, including their level of contribution to each project.

The commonality and degree of common contribution are defined below.

Given two projects *P* and *Q,* where **P** & **Q** represent the set of authors for each project, **R** is the set of common authors ($\mathbf{P} \cap \mathbf{Q}$) and $|\mathbf{R}|$ the number of authors in **R** (i.e. the number of authors in common):

$$commonality = (|\mathbf{R}| / |\mathbf{P}|) * (|\mathbf{R}| / |\mathbf{Q}|)$$

This measure provides an indicator of the proportion of authors common to two projects, regardless of their contribution to each project. The justification for this measure is, first, that contribution cannot only be measured in terms of credited source lines of code; second, that common authors may play a role in enhancing collaboration between the two groups of developers well beyond their direct contribution to the project source code.

Direct contribution is incorporated in a second measure. Given that $P_{\mathbf{R}}$ represents the contribution of all authors in the set **R** to the project *P*:

$$shared = P_{\mathbf{R}} * Q_{\mathbf{R}}$$

[42] See note 39. Developers who are linked together through this sort of clustering cannot be said to collaborate in the same way as do "co-participants" in a single project. Clustering would links two developers who work on no single project together but collaborate in different projects with a third common developer. Although these two developers are not direct collaborators, they do form part of a collaborating community through the interaction they have in common with the developer(s) that provide this link for clustering. Identifying the nature of collaboration in such communities helps to throw light on the development process, and may be supported with additional empirical evidence through the analysis of developer discussion groups. This assumption of a human or socialising element in collaboration between developers is key to the clustering model adopted, which essentially treats individuals common to multiple groups as forming links between these groups. An empirical analysis of the nature and composition of clusters over time (i.e. progressive release versions of software) could support this assumption, but it would seem to hold even for a static analysis given an understanding of the importance of discussion lists and other "socialising" interaction for the collaborative F/OSS development process.

In order to calculate a single value as the weight (w) of edges in the graph representing projects, these two attributes are combined. Two functions have been tested, and may suit different purposes. The first is a simple product:

*weight = commonality * shared*

As this results in the weight reducing rapidly as commonality and shared contribution reduce, sharpening differences between projects, a slight modification appears to make it more useful in identifying collaborations through this graph:

*weight = √(commonality * shared)*

Some properties of calculation of weight are:

1. the weight is a function of both the proportion of authors in common as well as the proportion of project code written by common authors
2. *commonality* is not biased towards the size of the author community. If all authors are common, this attribute will always be 1.0. If half are common, this will always be 0.25. Naturally, it is clearly biased to favour authors with a low contribution. This is both an obvious result as well as the reason for choosing to calculate commonality and shared contribution separately. They could be combined in one function that weighted common authors by their proportion to the total number of bytes and total authors, but that would assume a sort of continuum of authorship, rather than treating authors as discrete entities. This seems a good reason to provide a positive weight to authors as individual members of a team regardless of their contribution, in addition to the calculation of code contributed in shared contribution.
3. *shared* is not biased by relative differences in author contribution. I.e. if half of P and half of Q are written by the same authors, *shared* will always be 0.25 regardless of the number of authors or their distribution. If the distribution was, say, {0.4, 0.1} for P and {0.1, 0.4} for Q, a dot-product would return 0.08 although there's no difference, as far as we're concerned, between that author distribution and an equal – {0.25, 0.25} and {0.25, 0.25} – distribution, the only case where a dot-product would return the correct (for our purposes) result of 0.25.

6.4.1. Building clusters

In the second stage, the graph is analysed to identify vertices (projects) as potential cluster centres. Attributes useful at this stage include: the size of a project; the number of its authors; the number and strength of its links to other projects, by common authorship or by code dependency. For the purpose of identifying clusters with high levels of intra-cluster collaboration and relatively low levels of inter-cluster collaboration, starting from the best-linked projects – those with the highest number of and highest weighted links – is an obvious choice.

Once (some) potential cluster centres are identified, building a cluster around them is a fairly uncomplicated graph traversal problem – all links with a weight above a user-defined threshold are followed, and each visited vertex gets added to the list of projects belonging to the cluster. The authors on each traversed edge get added to the cluster of authorship. The "central" role of an author within a cluster is determined by his[43] relative contribution to projects within the cluster, or by how prolific a collaborator author is (i.e. the number of clustered projects to which the author is common).

This process is repeated, progressively identifying more clusters of authorship until all projects (and all their contributing authors) are placed within one or another cluster. It is important to note here that a cluster is not created as just a list of authors, but as a list of authors *and the projects they collaborate on.*

6.4.2. Analysing clusters: collaborators and non-collaborators

Within each cluster a clear and analytically useful structure appears: one based on a new measurement criterion that obtains through the graph traversal method, that of degree of collaboration. At its simplest, we see that there are collaborating authors – who are credited with authorship of more than one project – as well as non-collaborating authors, who are credited with authorship of only one project[44]. "Non-collaborating" authors do collaborate, of course, with other authors in developing that single project, but as we are trying to identify groups of collaborators, it is more useful to treat as collaborators only those who act as bridges between possibly distinct groups of people. The coherent delineator for such groups is the project that they work together on, hence this definition of "collaborating author" as one who contributes to more than one project, and therefore participates in more than one group of authors.

Following the graph structure, each cluster is built on the basis of collaborating authors who form the link between projects in the graph. Non-collaborating authors are added to the cluster by including all remaining authors for each project that has been linked into the cluster. Thus, if the set of authors for project *P* comprises collaborators and non-collaborators:

$$\mathbf{P} = (\mathbf{P}_{\text{collab}} \cup \mathbf{P}_{\text{noncollab}})$$

[43] F/OSS authors are almost 99% male, see BCG 2002 or FLOSS 2002

[44] There can be some inaccuracies in the underlying data that result in higher than actual numbers of "non-collaborators". This is because of situations where an author uses multiple identities to claim credit, and these identities may not be resolved in the data extraction process (manually or automatically). In some cases, these multiple identities may be used for separate projects (i.e. instead of a_1, a_2, a_3 appearing as joint authors of

The authors in $\mathbf{P}_{\text{collab}}$ will be included in the cluster **C** by virtue of being authors of projects other than *P* and thus appearing on edges in the graph. The authors in $\mathbf{P}_{\text{noncollab}}$ will be included in cluster **C** because project *P* gets included along with the set of its collaborating authors $\mathbf{P}_{\text{collab}}$ and therefore all the remaining authors of *P* are drawn into the cluster. The logic for this should be clear – $\mathbf{P}_{\text{noncollab}}$ are part of this collaborative cluster of authors although they only contribute to one project, because their co-authors in that project link them to groups of authors in other projects.

The simple distinction between collaborators and non-collaborators, although it does lead to interesting indicators of levels of collaboration for different groups of authors, can be made more complex by measuring the degree of collaboration for the "collaborators". Since a collaborator is an author who links two projects in a cluster, the number of links an author appears on is a simple measure of the author's degree of collaboration. Arguably, if authors are to be ranked within clusters (rather than within projects) based on their contribution, their degree of collaboration may prove to be more important than their contribution in bytes of source code. Indeed, preliminary clustering analysis of a number of projects shows that there isn't necessarily a strong correlation between high degrees of collaboration and high levels of source code contribution. When projects are looked at in detail (at the level of modules in the Linux kernel, say, rather than at a higher level where the entire Linux kernel is treated as a single "project") it often appears that small modules are written largely by developers with low levels of collaboration, while a number of highly collaborative developers contribute small parts to several different projects, tying those distinct groups of people together.

6.4.3. Clustering and dependency: cause and effect?

One reason for putting together information on clusters of authorship and dependencies between the packages they develop is to track "trade flows" among author clusters. As discussed in section 4, in the absence of monetary measures this helps answer the question "Who is doing how much of what with whom" – the clusters of authorship provide the (group) identities of the actors, while the underlying dependencies between the packages they collectively develop provides an understanding of the volume of their interaction.

However, since clustering analysis results not just in identifying groups of authors, but also links between packages (based on the existence common authors), there is a line of investigation possible into the possible correlation between common authorship and code dependency. Since data from both dependency and author clustering analysis can be attached to package pairs (the dependency links between any given two packages, and information on

projects *P, Q, R,* different identities appear for different projects, a_1 for *P,* a_2 for *Q* and so on). Thus, instead of one fairly "collaborative" author *a*, one could see multiple "non-collaborative" authors a_1, a_2, a_3 etc.

their degree of common authorship: *commonality, shared contribution*), it is possible to analyse the effect of dependency links and common authorship (and vice versa). Naturally, such effects may obtain only in future versions of the packages concerned, and the data is equally suitable for analysis across multiple source code versions[45].

6.5. Technical details: summary methodology and data structure

The data acquisition methodology is fairly specific to the structure and semantics of source code. There are several steps involved in acquiring the data, an overview of which is presented below (table 2).

6.5.1. *Table 2: Summary of stages of source code analysis and resulting data format*

Method	Explanation	Resulting data
Authorship credits	Heuristics for determining and assigning authorship of code segments at the file or package level.	List of the form {author, contribution in bytes of code} generated for each package
Duplicate file resolution	Many files are included in several packages, intentionally or by mistake. This results in double counting (a file is credited to its author multiple times, ones for each package where it occurs). Heuristics are used to resolve this problem and assign each file to only one package.	Corrected version of authorship credit list. List of shared files for each package.
Dependency identification	Files in one package may link to files in other packages. Heuristics are used to identify these links. With the addition of function-definition identification, a very accurate procedure results in identifying dependency links based where functions are uses and defined	For each code file in each package, a list of supporting files together with the packages they belong to. With the addition of function-definition identification, function names are also available.
Author clustering	Using a bi-directional graph structure where vertices represent packages and edges connecting them are weighted based on the existence and contribution of common authors, clusters of authors/packages are formed. The algorithm used does not split the graph into exclusive clusters, but only finds a single cluster given a central vertex. This allows flexibility in deciding how clusters are to be formed, based on varying edge-weight and distance thresholds, and allows the identification of overlapping clusters.	The author/project graph. Clusters of packages and the authors common to them, with information on "non-collaborators". For each project pair, data on commonality between projects, authors' degree of collaboration, number of common authors.

[45] Such analysis is being carried out as part of the LICKS project. See Ghosh & David 2003.

7. Conclusion, Outlook, and practical considerations

This paper has proposed methodology to extract, interpret and analyse empirical data from software source code. It describes an evolving methodology and tools in its current state, after having been tested and applied at various stages of development to diverse source code samples. Two projects that have used these tools are worth mentioning here: FLOSS and LICKS.

7.1. The FLOSS source code scan / Orbiten 2

The FLOSS project included a component (described in Part V of FLOSS 2002, intended as the 2nd Orbiten Free Software Survey) that applied many of the techniques described in this paper on a very large base of software, roughly 40 Gigabytes of compressed source code, i.e. approximately 3 billion source lines of code. Partly due to the scale of this code base, the analysis was carried at a fairly high level in that packages are rather large and not broken down into smaller sub-packages (the Linux kernel is treated as a single package, which means that dependencies or clusters are not identified for kernel components). Additionally, only current available versions were scanned, with no historical data or chronological analysis. Current analysis tools in the CODD/CODD-cluster suite are entirely non-interactive software and fairly technical – i.e. they are not user-friendly to operate and need programmer skills for customisation tasks. Clustering analysis does not provide graphical or visualization output, and there are at present no software tools as part of this project that perform chronological analysis. However, the development of such tools may not be necessary if it turns out that analysis of historical trends, say, is practical with the application of standard statistical analysis packages to data as currently generated. So far, this has seemed impractical – the difficulty of dealing with a graph of over 23,000 projects and 36,000 authors in a statistical package was the initial reason to develop customised methods and tools for clustering.

A preliminary evaluation of the methodology in practice must, however, be positive. Interesting results have been found in the dependency analysis, and a primary concern during cluster identification is the determination of appropriate threshold values to obtain useful results. It is perhaps unsurprising (but previously impossible to prove) that F/OSS projects are highly interconnected, so searching for a cluster centred with a zero threshold around the Linux kernel, say, tends to result in a huge cluster of authorship relative to the total code base. It will take some experimentation, together perhaps with visualisation techniques, to tailor the tools to generate clusters of manageable sizes that can be compared with one another as distinct groupings.

7.2. LICKS: Studying multiple versions of the Linux kernel

The LICKS project (Ghosh & David 2003) has looked specifically at three versions of the Linux kernel. Since this is a much smaller code base, it is possible to apply all the CODD tools in considerable detail (at the sub-package level, i.e. components of the Linux kernel rather than the Linux kernel as a single component in itself). It is also possible to apply the function-definition identifying techniques for accurate dependency analysis (as described in section 6.3.1, *Identifying function definitions as an aid to dependency analysis*) and integrate the resulting code dependency information with the clusters of authorship to determine the dependencies between distinct groups of authors, and identify correlations between dependency and authorship links.

If performed over multiple versions or over time, this analysis provides extremely interesting information on the exchange between groups, and could be a first step towards determining the internal economics of the functioning of F/OSS development. Aspects of participant development, migration, and reproduction become traceable.

For the first time, these methods point to the possibility of collecting concrete empirical data and analysis based on the source code – the only hard fact in F/OSS development – and extract the most of what is already ubiquitous, waiting to be studied. Empirical data extraction from source code should be of great interest to all social scientists, especially economists, but is also a valuable tool for developers to know about themselves and their organisation. This perhaps explains F/OSS developers' continuing interest in CODD and the Orbiten survey[46].

[46] The first CODD source code scan results were published online in late 1998 and received several hundred thousand hits in a few days, as did the first Orbiten Free Software Survey on its release in May 2000. This despite the fact that they provided only author contribution tables, and for a very small source code base.

8. Annexure: References and Literature

BCG (2002): Survey of free software/open source developers conducted by the Boston Consulting Group; see www.osdn.com/bcg

Boehm, Barry W., Software Engineering Economics, Prentice Hall, 1981. More details and updates at: http://sunset.usc.edu/research/COCOMOII/

Dempsey, Bert J, Debra Weiss, Paul Jones, and Jane Greenberg, "Who is an open source software developer?" Communications of the ACM. April, 2002. http://www.ibiblio.org/osrt/develpro.html

FLOSS: Free/Libre/Open Source Software Study, Rishab Ghosh, Ruediger Glott, Bernhard Krieger & Gregorio Robles, International Institute of Infonomics/MERIT, http://floss.infonomics.nl/report/

Ghosh, Rishab Aiyer (1996): Informal law and equal-opportunity enforcement in cyberspace, unpublished manuscript

Ghosh, RishabAiyer (1998): Cooking pot markets: an economic model for the trade in free goods and services on the Internet, in: First Monday, volume 3, number 3 (March 1998), http://www.firstmonday.org/issues/issue3_3/ghosh/index.html

Ghosh, Rishab Aiyer, and Vipul Ved Prakash, "Orbiten Free Software Survey", First Monday, volume 5, number 7 (July 2000), [http://www.firstmonday.org/issues/issue5_7/ghosh/]

Ghosh, Rishab Aiyer, "Cooking-pot markets and balanced value flows", in *Collaboration and Ownership in the Digital Economy*, Michael Century & Rishab Ghosh (ed.), MIT Press, *forthcoming* summer 2003

Ghosh, Rishab Aiyer and Paul David, "The nature and composition of the Linux kernel developer community: a dynamic analysis", 2003, SIEPR-Project NOSTRA Working Paper, draft available at http://dxm.org/papers/licks1/

Ghosh, Rishab Aiyer and Ved Prakash, Vipul (2000): The Orbiten Free Software Survey, in: First Monday, volume 5, number 7, http://www.firstmonday.org/issues/issue5_7/ghosh/

Grimm, Bruce T., Brent R. Moulton, and David B. Wasshausen, "Information Processing Equipment and Software in the National Accounts", NBER/CRIW Conference on Measuring Capital in the New Economy, April 2002, http://www.bea.doc.gov/bea/papers/IP-NIPA.pdf

Longstreet, David. 2001. Function Point Training and Analysis Manual. Longstreet Consulting Inc, Aug. 2001 http://www.SoftwareMetrics.Com/freemanual.htm

Raymond, Eric S., 1998, "The Cathedral and the Bazaar", First Monday, volume 3, number 3 (March 1998), http://www.firstmonday.org/issues/issue3_3/raymond/

Robles-Martínez, Gregorio. et al, 2001, "WIDI: Who Is Doing It?", Technical University of Berlin, http://widi.berlios.de/paper/study.html

Robles-Martínez, Gregorio, Jesús M. González-Barahona, José Centeno González, Vicente Matellán Olivera, and Luis Rodero Merino, "Studying the evolution of libre software projects using publicly available data", 25th International Conference on Software Engineering, 2003, http://opensource.ucc.ie/icse2003/

Tuomi, Ilkka, "Evolution of the Linux Credits File: Methodological Challenges and Reference Data for Open Source Research" – working paper, 2002, available at http://www.jrc.es/~tuomiil/moreinfo.html