



Dynamics of Institutions and Markets in Europe is a network of excellence of social scientists in Europe, working on the economic and social consequences of increasing globalization and the rise of the knowledge economy.
<http://www.dime-eu.org/>

DIME Working Papers on **INTELLECTUAL PROPERTY RIGHTS**



Sponsored by the
6th Framework Programme
of the European Union

<http://www.dime-eu.org/working-papers/wp14>

Emerging out of DIME Working Pack:

'The Rules, Norms and Standards on Knowledge Exchange'

Further information on the DIME IPR research and activities:

<http://www.dime-eu.org/wp14>

This working paper is submitted by:

Rishab Ghosh^{*} and Paul David^{}**

^{*}UNU-MERIT

^{**}Stanford University, All Souls College Oxford and UNU-MERIT

*Relating social structure to technical structure:
findings from the Linux kernel*

**This is Working Paper
No 77 (May 2008)**

The Intellectual Property Rights (IPR) elements of the DIME Network currently focus on research in the area of patents, copyrights and related rights. DIME's IPR research is at the forefront as it addresses and debates current political and controversial IPR issues that affect businesses, nations and societies today. These issues challenge state of the art thinking and the existing analytical frameworks that dominate theoretical IPR literature in the fields of economics, management, politics, law and regulation- theory.

Relating social structure to technical structure: findings from the Linux kernel

Rishab Ghosh¹ and Paul David²

May 2008

Introduction

New contributors³ to free software development appear to be socially influenced in their choice of first contribution, preferentially joining modules with a large number of existing developers. This leads one to suggest that social links between modules in the form of common developers may be associated with technical links between modules – for example, in the form of functional dependencies. This paper analyses data gathered from a detailed study of the structure and composition of the Linux kernel developer community, as sampled through three versions of the Linux kernel. The relationship between the degree of common authorship of module pairs is compared with their degree of functional inter-dependence, and tested for causality over three versions of the Linux kernel. Open source projects demonstrate an apparent ability to manage the development of complex, growing codebases more efficiently than the software engineering literature predicts is possible for standard software development (within firms). This paper provides some initial evidence to suggest that this efficiency may be supported by a combination of structural and social factors that make open source inherently suitable for skills transfer and distributed management.

The code base used for this analysis is the Linux kernel. Three versions of the software were used: version 1.0, version 2.0.30, and version 2.5.25. The use of three versions with a significant time interval separating them allows meaningful analysis to be performed on the dynamics of changes within the development of the kernel. This paper is based on data extraction and analysis that was first reported in Ghosh & David 2003.

An overview of the three versions can be found in Table 1.

¹ UNU-MERIT

² Stanford University, All Souls College Oxford and UNU-MERIT

³ Support for this research was provided by the Project on the Economic Organization and Viability of Open Source Software, which was funded under National Science Foundation Grant NSF IIS-0112962 to the Stanford Institute for Economic Policy Research. Further support was provided by the European Union's FP6 project FLOSSMETRICS.

Why the Linux Kernel?

The Linux kernel is at the core of the GNU/Linux operating system, and is one of the main factors behind the current success of open source software. It also has the advantage of being a clearly defined (and bounded) piece of software with historical versions available in a continuous progression since at least 1993.

The Linux kernel is a large project by any measure. The most recent version studied in the LICKS project had over 3 million lines of code and almost 2,300 identified developers. Moreover, due to the nature of the kernel, which provides the core services of the operating system (such as memory management, input/output, filing systems, networking, interfacing with various devices) it must be able to function in a well integrated fashion in order to run at all. The open source development mode does not require the integrated nature of code functioning to be directly represented in a tight structure for developer collaboration, but it does require a strong degree of collaboration on a large scale. Furthermore, the technical structure of the Linux kernel is well understood, and the delineation of modules required in order to make a reasonable map of functional dependencies is well defined.

The Linux kernel a typical open source project we believe it is safe to say that there are no typical open source projects *per se* at any rate, we are only beginning to collect the sort of empirical data on a sufficiently large and diverse set of projects in order to classify projects by type⁴.

The methodology used in the analysis is described in detail in Ghosh 2003a, and its specific application to the Linux kernel in Ghosh & David 2003. It depends primarily on the CODD suite of psoftware-analysis tools⁵. To summarise, the analysis:

- examined the source code of each version of Linux, scanning files for claims of authorship and crediting groups of files collected as modules to the respective authors
- identified dependencies between source code modules, by identifying functions defined in and called from each file and grouping them by module.
- identified links between modules by identifying authors in common ("social links"), and links between modules based on the technical dependencies ("technical links").

⁴ This is reported on in the FLOSSMETRICS project. see flossmetrics.org

⁵ CODD, first released in 1998, was designed by Rishab Ghosh and Vipul Ved Prakash, implemented by Vipul Ved Prakash and Gregorio Robles. CODD-Cluster and CODD-dependency were designed by Rishab Ghosh and implemented with Gregorio Robles.

Contributor concentration and "co-participation"

A general rule with almost fractal properties for any collection of open source software appears to be that most modules are small and that most contributors have contributed to very few (usually one) module. Ghosh & David 2003 shows this pattern for the Linux kernel, where a module is a group of files within the kernel; a similar pattern was first observed in the 2000 Orbiten survey⁶ and the FLOSS Source Code scan⁷ where the entire Linux kernel was treated as a module (or "package" as termed by those studies). The observation that most packages are small, or developed by small groups of authors, has since been documented in various studies⁸.

It is a cliché that FLOSS has a collaborative development model. How collaborative is it, really? The mythology of open source the bazaar, many-eyeballs suggests all projects are results of massive collaboration. We do not need to show again how this is simply not true for the majority of projects. Data from the Orbiten and FLOSS surveys, and again from Ghosh & David 2003, show that most modules have one or two developers, or a few more at most; the same data show too that most developers contribute to a handful of modules.

Some researchers have suggested that such data require a reassessment of our understanding of the FLOSS model as one of massively collaborating development, especially given that most development occurs in tiny groups. Although there is merit in criticizing the many-eyeballs hype as overly simplistic, the suggestion that most development occurs in small groups without large-scale collaboration implies that FLOSS projects develop in a hermetically sealed environment (the isolated "caves" of Krishnamurthy) much as proprietary software development often does. Indeed, if one observed a proprietary software project being developed by a team of 3 individuals, one would be correct to assume that only those 3 collaborated in completing that project.

The suggestion that most FLOSS development activity occurs in small groups also seems to contradict the motivations presented by developers themselves for participating in FLOSS communities the most important reasons given invariably involve the learning of new skills or the sharing of knowledge⁹.

⁶ Ghosh & Ved Prakash 2000

⁷ Ghosh et al 2002 Part V

⁸ Krishnamurthy 2002, Healy & Schussman 2003

⁹ Ghosh et al 2002 Part IV, Survey of Developers. See also Ghosh and Glott 2005 on skills and FLOSS

Since developers are only human, and usually devote only part of their FLOSS development time to any single project, they may feed in results from their collaboration with other developers in external FLOSS projects. Evidence of such cross-project collaboration (or extra-project collaboration) has largely been hard to find, and has often been assumed to result from the widespread informal communication systems that FLOSS relies upon (specifically e-mail discussion lists). However, we are able to show at least for the Linux kernel how authors collaborate in terms of actual code production, and do so extensively. Note that 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. There is a strong argument that co-participation in itself implies a high degree of collaboration in the FLOSS arena (unlike in publishing, where joint authors of a paper could possibly have contributed different sections with no collaboration at all). For a computer program at the level of a single file or source code module, collaboration in the form of awareness of other developers' contributions is a pre-requisite in order for the program to function at all.

As has been noted above, most authors contribute to only a single module. However, Figure 1 shows that most authors in the Linux kernel collaborated with several developers. Indeed, there are few authors in any of the three versions of the kernel studied who wrote *only* one single module all on their own, as almost all have at least one collaborator¹⁰.

True, many developers contribute to only one module, and many modules have only one developer. However, it turns out that *most developers who contribute to just one module choose to contribute to a large one* i.e. one with several developers. This explains the fact that most (66%) developers in version 1.0 have collaborated with between 21 and 50 others, and 69% of version 2.0.30 developers have collaborated with between 51 and 200 others. The fact that single-module contributors tend to contribute to modules with many developers suggests that reputation, learning or simply social group-formation does indeed attract new developers to major modules.

¹⁰ From our findings, the number of developers with no collaborators was 0, 1 and 2 for versions 1.0, 2.0.30 and 2.5.25 respectively. It should be noted that the CODD accreditation method rarely provides false positives, in that any two authors identified as co-developers would have credit claims on a common module, which means they would have contributed to the module though not necessarily at the same time. The exception to this is when two developers are actually the same individual with different identities in the source code files something for which CODD allows correction, but cannot do fully automatically. Much more likely as can be seen from Ghosh & David 2003 and David et al 2005 on unsigned code is *false negatives*, where CODD identifies fewer developers (and hence co-developers) than actually exist. Given that the true number of co-developers are thus likely to be higher than represented by our data, it would be reasonable to argue that *no* authors contributing to the Linux kernel versions studied have done so with absolutely no co-participation with other authors.

Note that this co-author value is exactly the same as the centrality degree of the contributor in social network analysis — if contributors are nodes on a graph and connected to all others with whom they have a module in common. In order to compare across versions, it is useful to measure not the actual number of co-authors, but the co-authors as a share of the total number of authors. The cumulative distribution of this co-authorship share is shown for all three successive versions of the Linux kernel in Figure 2.

As the figure shows, co-authorship rises rapidly and roughly 90% of authors are directly connected to at least 5% of other authors; 60% are connected to at least 10% of co-authors. Nevertheless, this slows down logarithmically, so very few authors (only the top 10%) are connected through co-authorship with more than 20% of the total authors.

Notably, the distributions of co-authorship share are remarkably similar despite the enormous increase in the number of authors from 158 to 618 and 2263 over the three Linux kernel versions. As the number of total authors increases, the distribution becomes more skewed towards lower co-authorship shares among the lowest quintile. But not very much. In fact, the interquartile ratio between the top and bottom quartile increases logarithmically in proportion to the increase in total authors. This suggests that while co-authorship shares do decline (the community becomes more loosely-knit) as the number of total authors increases (the community grows), this loosening happens at a much slower rate than the community's growth rate. And the loosening slows to a near halt. Thus there is a certain level of tightness in the community that is not lost even as a community grows rapidly in size. This is an argument in favour of rapid growth of software projects in FLOSS mode, and demonstrate how the cooking-pot market of FLOSS allows groups of people to effectively sustain collaborations at very high levels of growth in complexity.

Interestingly a similar argument has been made from a software engineering perspective, for sustainable near-linear or even super-linear growth in FLOSS projects, in contrast to the widely held, empirically validated laws that software projects must have sub-linear growth! We return to this issue in our conclusions.

Another feature that is noticeable in Figure 2 is a sort of heaping, as the cumulative distribution increases in steps. This is actually a result of the fact (common to social network analyses of FLOSS) that any author of a module is considered connected (or co-author) with all other authors to

¹¹ Robles 2006 (pp. 86-100) demonstrates, following Godfrey & Tu 2000, that several FLOSS software projects achieve linear and even super-linear growth over time, violating Lehman's Laws of Software Evolution (Lehman et al 1997) which were empirically validated by studying proprietary software development over several decades.

that module. Together with the fact that developers' first contributions tend to be to large modules, there is a high number of single-module developers who will have exactly the number of co-authors as total authors for such particularly large, popular modules. As the total number of modules increases from version to version (from 30 modules in version 1 to 169 modules in version 2.5.25) this effect gets smoothed out.

Where developers allocate their contribution

There are many things that may influence dispersion¹² of developer contribution across modules, but in general there appears to be a negative correlation¹³ between dispersion and total number of modules contributed to — as authors contribute to more modules, they tend to spread their contribution more evenly across them. On the other hand there is a positive correlation between dispersion and mean contribution (in bytes) to modules — as an author's mean contribution per module increases, the contribution tends to be more concentrated in a few modules.

There is, however, *no correlation between dispersion and total author contribution* in bytes, which suggests that there may be two categories (at the extrema) of equally productive authors, the first including authors who tend to spread their contribution across several modules evenly, the second of authors who tend to concentrate their contribution in a few of the modules in which they are involved.

Whether this is a result of a conscious decision of developers to focus most of their contribution on a few modules or spread their code across modules (possibly focusing on specific tasks that are useful for a number of modules) is a subject for further research.

It is arguable that developers who focus on specific modules and those who distribute their contribution more evenly may have different incentives — though it is not necessary that concentrating on a few modules (leading to a high-profile in those modules) must lead to a better reputation than spreading oneself across several modules (leading to a wider reputation possibly for specific, in-demand types of coding that are needed in several modules).

It is also likely — perhaps not for the Linux kernel alone, but in general — that these two groups reflect different skills development patterns, especially when one remembers that developers contributing to few modules tend to contribute to large, popular ones. Thus, developers who

¹² The dispersion measure used here is, for each author, the variance of the author's contribution to individual packages measured as a fraction of the author's contribution to all packages. Packages with no contribution from the author are excluded.

¹³ For v2.5.25, Pearson two-tailed correlation coefficient of 0.203 between mean contribution and dispersion; -0.259 (negative) correlation coefficient between number of modules contributed to and dispersion

contribute extensively to a single module may have joined that popular module as relative novices and rapidly improved their skills within that module, thus extending their contribution to it.

Developers who contribute to many modules are more likely to have had (relatively more) extensive skills to begin with, whether learnt from participation in other projects, or from outside the FLOSS community. This would allow them to initiate (and often complete) modules all on their own or with little support from others.

Dependency information

Software is by nature collaborative in functioning and software modules usually depend on features and components from several other modules. Such dependencies must be explicitly detailed in a way that they can be determined automatically, in order for an application to run. We identified these dependencies in some detail, using the function-definition identification method described previously. The scope of this analysis was very large: to illustrate, dependency analysis for the Linux kernel version 2.5.25 alone generated 600Mb of data, identifying over 5 million function dependencies for some 50,000 functions defined across more than 12,000 source code files (about 175 Mb of source code). This was then summarised to 8,328 dependencies between 178 modules.

An analysis of the distributions between supporting and depending modules shows that most modules depend on a small subset of highly supporting modules, while all modules tend to depend on a more-or-less equal number of other modules, as shown by the Gini coefficients for support and dependence in Table 2.

Are social structure and technical structure related?

New contributors to free software development appear to be socially influenced in their choice of first contribution, preferentially joining modules with a large number of existing developers. It appears that social links between modules in the form of common developers may be associated with technical links between modules.

The analysis of common authorship between modules is based on building a graph somewhat different from a social network analysis: instead of linking authors based on common modules, we link modules (of the Linux kernel) based on common authorship¹⁴. I.e. if two modules have at least one contributor in common, there is a link between them. There are a number of ways of

¹⁴ This approach is useful for two reasons: first, linking authors if they have both contributed to the same module results in all contributors to a single module being linked to each other, thus providing no more information than a graph where modules rather than authors are nodes. Second, technical dependencies are between modules, not authors, clearly, so this approach allows one to relate social and technical dependencies as the nodes are in both cases modules.

determining the strength of this link between a pair of modules¹⁵: the number of common authors; the ratio of common authors to the total authors of each module pair (*commonality*); and the ratio of the code contributed by these common authors in the two modules to the total code of the two modules (*shared* ratio). This results in *undirected* weighted graph where modules are nodes, and the *social links* between them are the edges weighted by three different measures of connectivity strength. The graphs were created for each of the three versions of the Linux kernel studied, using the CODD-Cluster tool¹⁶.

Technical dependency was identified at the function level, resulting in a *directed* weighted graph where modules are nodes and two modules are linked if one defines functions that are called in the other. Two parallel directed edges are made, one from the supporting module defining functions to the dependent module calling functions, and one in the opposite direction. It is possible, though not common, for two modules to call functions defined in each other (co-dependency), in which case a second set of directed edges would exist between the pair.

The edges in opposing directions are symmetric, so they are equally weighted, by the number of functions defined in one module and *used* in the other¹⁷. Distributions of these dependencies across modules in the Linux kernel are described in Ghosh & David 2003.

Common authorship, or social links

For the analysis in the Linux kernel, the graph was treated as a dataset of all possible pairs with numerical variables indicating the three separate weights: number of connectors (LRN), *commonality* (LRC) and *shared* or *co-authored code* ratio (LRS).

It turned out that *commonality* was positively correlated to the absolute number of connectors¹⁸, but

¹⁵ This methodology was detailed in Ghosh 2003.

¹⁶ *supra note 5*

¹⁷ *Used* because in fact, the measure is functions defined in one module and *declared* in the other. Declarations are intents to use, but not necessarily actual function calls. Since such declarations are typically from *header files* that are *included* in the dependent module (see section), they may declare functions that are not actually used. Moreover, the number of functions declared depends on programming style and conventions if more are consolidated in a single header file, they will all be automatically *declared* when the header file is *included* by a dependent module. Besides, the number of functions is not directly related either to their size, or their importance. A more accurate measure of the extent of dependency would actually count the number of functions used. However, there is really no end to the level of detail possible in search of greater accuracy: should the functions used be weighted by the number of times they are called? By the size of the functions (which is extremely variable)? By some indicator of the importance of the function (for which size is not necessarily a proxy)? And so on. Each level of detail adds enormously to the complexity of the algorithm and greatly reduces the scale at which it can be applied.

¹⁸ Pearson 2-tailed correlation significant at the 0.01 level, 0.312, 0.567 for Linux kernel versions 2.0.30 and 2.5.25 respectively; for version 1.0 there was no significant correlation.

negatively correlated to the number of total contributors for each module pair¹⁹. This is despite the significant *positive* correlation between the number of connectors and the number of total contributors for each module pair²⁰ - obviously because the number of total contributors grows far more rapidly than the the number of connectors.

Looking at LRN, it is unsurprising that a large majority of links are due to a single author, accounting for 82%, 66% and 52% for Linux kernel versions 1.0, 2.0.30 and 2.5.25 respectively. LRC provides a measure of how the presence of tightly integrated teams in the modules, and it is striking that the top quartile of most-connected module pairs have at least 25% of their combined developers in common in version 1.0, and 10% in version 2.0.30. This reduces greatly to at least 2% developers common to the top quartile of most-connected module pairs in version 2.5.25, reflecting perhaps a dispersion of teams across the module space¹.

The shared or co-authored code ratio is the code contributed to the two modules by the authors they have in common, expressed as a share of the total code for the two modules, for every module pair. These are negatively correlated to the number of developers as well as the size of the modules. But unlike LRN, both LRC and LRS have log-normal distributions.

As with LRC, this measure of commonality (LRS) declines rapidly over successive versions, with common authors contributing at least 29% of total code for each of the top quartile of most-connected module pairs in version 1.0, down to 14% of total code in version 2.0.30 and finally only 4% in version 2.5.25. This decline across versions is slower than for commonality, indicating that common authors become more productive in terms of code output over time²². Overall, as Figure 3 shows, this measure is closely correlated across different versions on Linux.

Function calls, or technical links

For this analysis in the Linux kernel, identifying the relationships between social and technical links, the weights on the technical links graph were not used due to uncertainties about the

¹⁹ Pearson 2-tailed correlation significant at the 0.01 level, -0.411, -0.253 and -0.119 for Linux kernel versions 1.0, 2.0.30 and 2.5.25 respectively.

²⁰ Pearson 2-tailed correlation significant at the 0.01 level, 0.136, 0.253 and 0.276 for Linux kernel versions 1.0, 2.0.30 and 2.5.25 respectively.

²¹ And, perhaps, a result of people ending their contribution over the 8-year interval between the three versions studied. Note that across versions, the change in commonality declines faster than the change in code size or total author numbers increases.

²² No implication for causality is made here - it could be that common authors become more productive as they work with teams across modules; or that more productive authors are more likely to work across modules in successive versions. Some of this may also be an artifact of the CODD data collection method, since may record *cumulative* code contribution by authors.

interpretation and distribution of the weights²³. Moreover, since the social links graph was bi-directional, the technical graph was also converted to a bi-directional graph. As a result the technical graph actually used for this correlation exercise was converted to a bi-directional unweighted graph equivalent to a dataset of all possible module pairs with a Boolean variable that is true if a technical dependency link exists between the pair in either direction.

Social and technical effects

As described in the previous section on co-participation, most modules have just a few authors, and most authors have contributed to just one module. These lone contributions are not made to 1-author modules: new contributors choose modules with many other contributors. As the histogram in Figure 4 shows, the mean co-developers per module²⁴ is rather high for 1-module developers, over 45% have contributed to a module with more than 200 co-developers. For developers contributing to 2 modules, the mean co-developers is not much less, indicating that the 2nd module contributed to is not much smaller. This indicates a continuing preference for contributing to large modules. Eventually, contributors must run out of large modules, but they start contributing to smaller modules only if they contribute to many modules.

A developer's mean number of co-developers per module is negatively correlated to the total number of modules to which he has contributed²⁵. However, contributors don't necessarily know how many co-authors they have or will have for a given module, since the number of developers is not always apparent. But code size of module is easily determined and is tightly correlated to the number of authors²⁶. Size is thus an explicit proxy for popularity. Thus, developers are attracted to popular modules, partly (we assume) because they have many other developers. We believe this may be linked to the skills transfer effects, i.e. that developers with lower skills find it easier to contribute, initially, to modules with large numbers of people who could potentially support them. Writing a module on your own, or with fewer people, requires higher skill levels especially if it is to be of a quality good enough to be acceptable to a broader FLOSS community.

For a closer look at the technical and social relationships between modules, we can explore the correlation between the Boolean variable indicating a presence of a technical link in a module pair,

²³ *Supra note 8*

²⁴ The ratio of total number of co-developers including duplicates by the total number of modules

²⁵ Pearson 2-tailed correlation significant at the 0.01 level, -0.225, -0.222 and -0.141 for Linux kernel versions 1.0, 2.0.30 and 2.5.25 respectively.

²⁶ Pearson 2-tailed correlation significant at the 0.01 level, 0.890, 0.892 and 0.894 for Linux kernel versions 1.0, 2.0.30 and 2.5.25 respectively.

and the variables indicating common authorship. There is a strong degree of coincidence between the presence of an author link and a code link. There is a significant correlation between strength of author link as measured by the number of common authors (LRN) and presence of code link. This is weak for version 1.0 (Spearman ρ of 0.130 significant at the 0.1 level) but stronger for version 2.0.30 (Spearman ρ of 0.241 at the 0.1 level) and version 2.5.25 (Spearman ρ of 0.353 at the 0.1 level of significance). Similar correlations exist between the presence of a code link and the other measures of strength of the social link (commonality and co-authored code share).

There is clearly already a technical effect of this social phenomenon: it encourages the power-law distribution of module size and module authorship (for which there is no *technical* reason, per se; indeed, as our conclusion will show, the software engineering literature suggests a preference for *smaller* modules which would imply a more even distribution of module size). The social result of this agglomeration of authorship may be a skills transfer linked to co-development.

Indications of the intertemporal impact of code links on future author links are present, though weak²⁷, and indications of the intertemporal impact of author links on future code links are similar. So it is not possible to identify the direction of causality (the data is not sufficiently granular in addition to the artifacts of CODD, which often records *cumulative* contribution over versions, the time difference between versions is very large, several years).

Code links could lead to social links, if developers who use functionality of other modules eventually start examining those modules and improving them as their own skills and needs grow.

On the other hand, social links could lead to code links, if developers who work on common modules start making these modules more technically dependent on each other.

The direction of causality is rather important if code links lead to social links, that provides a direct basis for how technical skills development takes place through learning-by-doing in FLOSS communities, as empirically proven from the perspective of developers and employers by Ghosh and Glott (2005, 2005b). However, if social links lead to code links, since social links are clearly expanding that may lead to a dangerous level of technical linkage that is not technically necessarily. Indeed, this would work against the modular nature of code that supposedly allows the open source development process to function relatively well, and lead to spaghetti code with

²⁷ Spearman ρ of 0.252 at the 0.01 level of significance for the presence of a code link in version 1.0 affecting the strength of the author link (number of common authors, LRN) in version 2.0.30. Spearman ρ of 0.214 at the 0.01 level of significance for the presence of a code link in version 2.0.30 affecting the strength of the author link (number of common authors, LRN) in version 2.5.25. For the strength measured by co-authored code share (LRS) the correlation is 0.300 and 0.185 across the first and second, and second and third versions respectively.

technical links increasing the complexity of code to possibly unsustainable levels. Some scholars have argued that this is already happening (Schach and Offutt 2002), though the continued success of large FLOSS projects may indicate that this is not yet a threat, or that this is in fact not happening.

Conclusions

Some studies have suggested that FLOSS development models allow for greater productivity (per developer hour worked) than traditional models of software development within firms²⁸. Indeed, this is a matter of much discussion within the software engineering community. FLOSS developers appear to work independently, optimising their level of contribution with relatively low coordination costs. The costs of developing software increase exponentially in most models of proprietary software development, apparently largely due to coordination costs – the larger the code base, the larger the team size and the greater the complexity. Empirical findings suggest that FLOSS projects appear to violate many laws that state that software projects must have sub-linear growth, assuming that effort grows exponentially with size, and thus – since effort cannot realistically grow exponentially, size cannot grow linearly within a given project. The exponentially growing complexity of large projects is also a reason why one might normally expect more projects of a similar, moderate size, rather than the distribution found in FLOSS (fractally, at all levels of "project") of many very small projects and a few very large ones.

However, evidence from the software engineering literature shows that many FLOSS projects do have linear or super-linear growth²⁹, suggesting a structural shift in the level of complexity that can be managed.

One hypothesis, made in Ghosh et al 2007, is that although the coordination and management skills required for large groups of people to develop FLOSS is significant, centralised coordination effort (and coordination costs) appear to be missing or significantly reduced. This may be because the huge numbers of developers are not coordinated as a single team. Several studies, including social network analysis suggest, that FLOSS developers work independently in a highly modularised self-organising structure³⁰. This substitution of complex coordination with complex organisational structure may require a wider distribution of coordination abilities and coordination systems, but less *high-level* coordination.

²⁸ Ghosh et al 2007, "7.2 Primary production of FLOSS code"

²⁹ See e.g. Godfrey and Tu 2000; Succi et al 2001; Robles et al 2005; Koch 2005.

³⁰ See e.g. Crowston and Howison 2005; Wendel de Joode and Kemp 2001; Trung and Bieman 2005; Mockus et al 2002.

Ample evidence exists to suggest the strong role of FLOSS developer communities in the development of skills, especially including management, team work and coordination skills³¹ - and this skills development potential is a clear motivator for a continuous stream of new entrants to the pool of FLOSS developers. However, there is little formal process for the transmission of these skills; they "rub off" onto lower-skilled developers through proximity to higher-skilled developers - and the software the latter write.

We have shown in this paper that the self-organisation nature of FLOSS projects is not limited to their functional and government structure. FLOSS projects also self-organise a system for skills transfer that ensures their own sustainability. This system is driven by the tendency of (relative) novice developers to adhere to groups of developers (and code) that are large. In such a situation, there is a real, valuable contribution made by novice developers (and contribution is essential to any system of "learning by doing"). However, this contribution is small in proportion to the code base and developer community of a large module, and the potential skills transfer in the form of close interaction³² with more skilled developers proportionately large.

We have shown in this paper that there is a correlation between technical links in the form of functional dependence between modules and social links in the form of co-participation in the development of modules. Such a correlation supports the notion that skills transfer is an emergent property of the inherent structure of FLOSS communities; skills transfer is nothing if not a relationship between the technical, in the form of knowledge, and the social, in the form of implicit mentorship. While our data does not allow us to identify a degree of causality from functional dependency to co-participation, it *does* show that certain technical features - the presence of a large codebase in a module - cause social links in the form of co-participation for first-time contributors.

If social links result in technical links, or there is no causality between them, the explanation for how the system and process for skills transfer works may have to rely on other attributes of FLOSS communities. If technical links in the form of functional dependencies do result in social links in the form of co-participation, and hence potentially also skills transfer, it would suggest the development of a sophisticated, if self-emergent, system for skills transfer based on technical association. One could argue that people start with using a function; then examine how it works; then modify it, thus collaborating with (and potentially learning from) its previous authors. Further research can help

³¹ Ghosh & Glott, 2005.

³² In the sense of "working with people by working with code", not necessarily actual communication with other developers; we have argued previously that when it comes to software, this is indeed interaction.

identify the particular skills transfer mechanisms involved, and the interplay between technical and social relationships in FLOSS software projects.

Table 1: Linux kernel code base overview

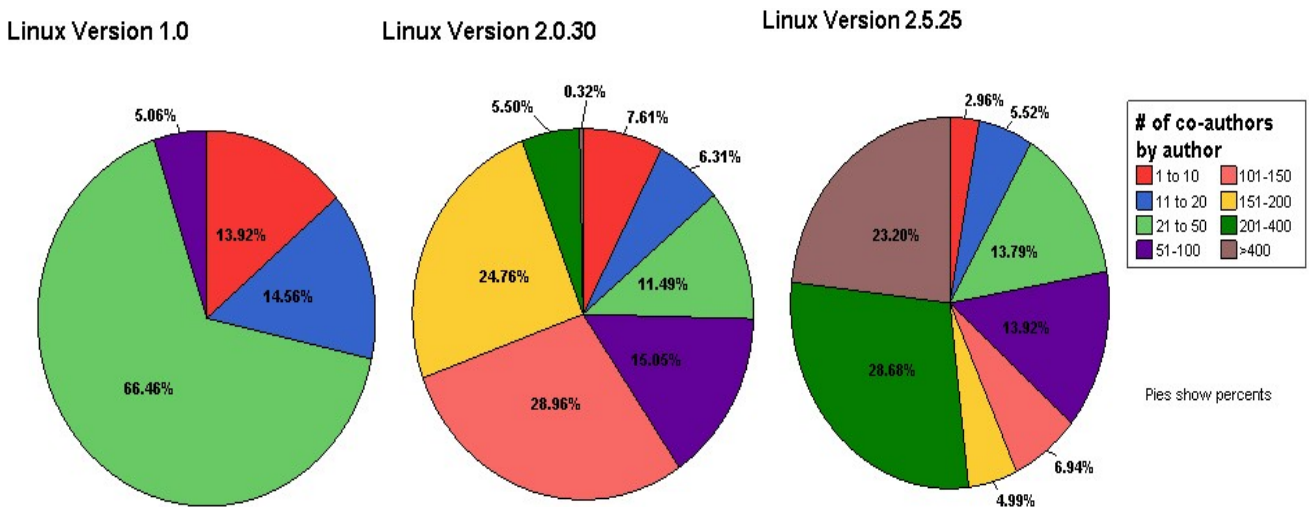
Linux kernel	Version 1.0	Version 2.0.30	Version 2.5.25
Approximate release date	Mar-94	Apr-97	Jul-02
Number of modules [^]	30	60	169
Number of files	593	2,155	12,451
Number of authors [*]	158	618	2,263
% of code uncredited [*]	18.8%	12.2%	14.9%
Number of defined functions [*]	1,748	7,808	48,006
Physical source lines of code ⁺	121,987	537,773	3,157,543

Notes: [^] as defined for this study; ^{*} as identified by CODD; uncredited code is code for which CODD was unable to find any author signatures. ⁺ as identified by SLOCCCount³³.

Table 2: Gini coefficients of support and dependence

Linux version	Support	Depend
1.0	0.56	0.35
2.0.30	0.44	0.20
2.5.25	0.66	0.15

Figure 1: Collaboration with other authors, v1.0, v2.0.30, v2.5.25



³³ see Wheeler 2001

Figure 2: Cumulative distribution by author of co-authorship share, all Linux versions

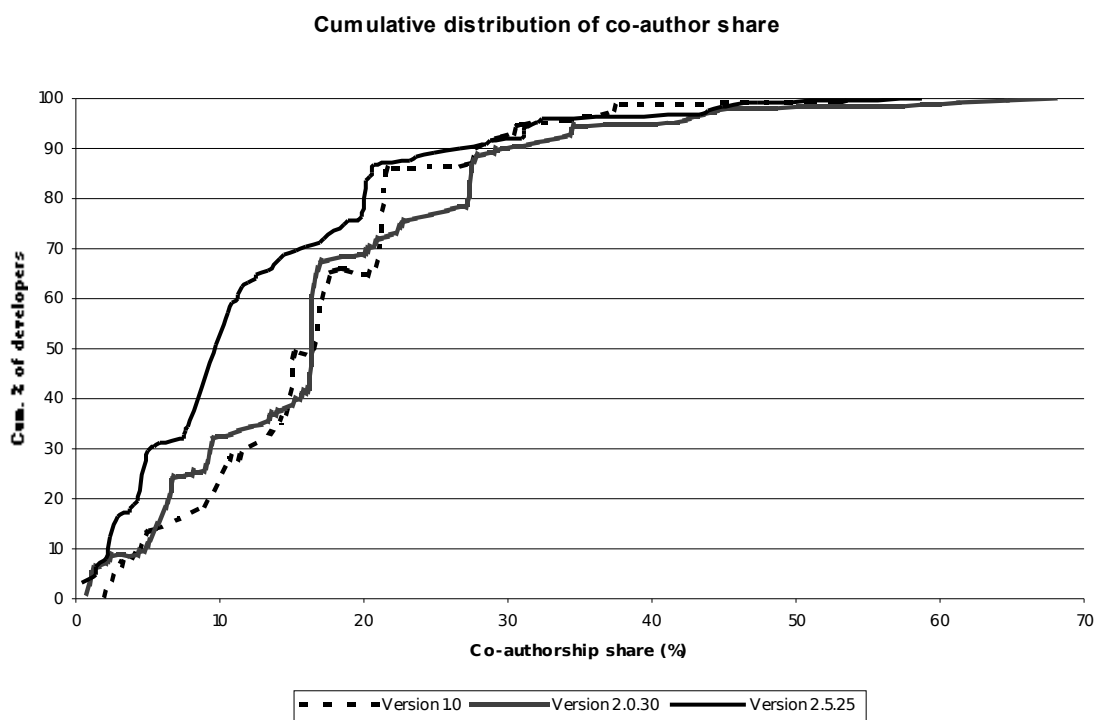
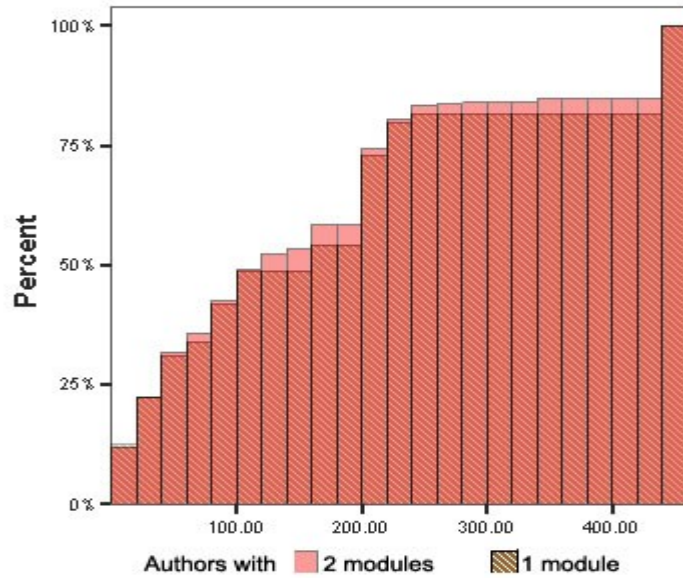


Figure 3: Co-authored code ratio is related across versions of Linux

Proximity Matrix

	Correlation between Vectors of Values		
	LRSV10	LRSV20	LRSV25
LRSV10		.926	.835
LRSV20	.926		.864
LRSV25	.835	.864	

Figure 4: Mean co-developers per module, for 1 and 2-module authors, v2.5.25



Cumulative histogram showing mean co-developers per module for 1-module authors and 2-module authors.

References

- Crowston, Kevin and Howison, James. 2005. "The social structure of free and open source software development", *First Monday*, volume 10, number 2 (February), http://www.firstmonday.dk/issues/issue10_2/crowston
- David, Paul A, Dalle, J-M, Ghosh, R.A. & Wolack, F.A. 2005. Free & Open Source Software Development and the Economy of Regard , Proceedings of the Workshop on Open Source Software and Intellectual Property in the Software Industry. *3rd Bi-annual Conference on The Economics of the Software and Internet Industries* (Toulouse, France: Centre for Economic Policy Research & Institut d Economie Industrielle). January 20.
- Boehm, Barry W.. 1981. *Software Engineering Economics*, Prentice Hall. More details and updates at: <http://sunset.usc.edu/research/COCOMOII/>
- Dempsey, Bert J, Debra Weiss, Paul Jones, and Jane Greenberg. 2002. A Quantitative Profile of a Community of Open Source Linux Developers, *Communications of the ACM*. April. [<http://www.ibiblio.org/osrt/develpro.html>]
- Ghosh, R.A., Glott, R., Krieger, B. & Robles, G. 2002. *FLOSS: Free/Libre/Open Source Software Study*. European Commission / MERIT, <http://flossproject.org/report/>
- Ghosh, Rishab Aiyer and Paul A. David. 2003. The nature and composition of the Linux kernel developer community: a Dynamic Analysis, SIEPR-Project NOSTRA Working Paper (21st February). Available at: dxm.org/licks
- Ghosh, R.A. & Ruediger Glott 2005. The Open Source Community as an environment for skills development and employment generation .*European Academy of Management (EURAM) Conference*, Munich, May 4-7.
- Ghosh, R.A. & Ruediger Glott 2005b. FLOSSPOLs: Skills Survey Interim Report *European Commission / MERIT*. September. Available at: www.flosspols.org/deliverables.php
- Ghosh, Rishab Aiyer, and Ved Prakash, Vipul. 2000. Orbiten Free Software Survey *First Monday*, Vol 5, No. 7 (July) [available at http://www.firstmonday.org/issues/issue5_7/ghosh/]
- Ghosh, R. A. 2003. Clustering and dependencies in free/open source software development:

- Ghosh and David. *Relating social structure to technical structure: findings from the Linux kernel*.
Methodology and tools, *First Monday*, volume 8, number 4
- Ghosh, Rishab Aiyer et al. 2007. *Economic impact of open source software on innovation and the competitiveness of the Information and Communication Technologies (ICT) sector in the EU*. European Commission. Available at: www.flossimpact.eu.
- Godfrey, Michael and Tu, Qiang. 2000. Evolution in Open Source Software: a case study, in *Proceedings of the International Conference on Software Maintenance*, pp. 131-142. San Jose, California
- Healy, Kieran and Alan Schussman. 2003. The Ecology of Open Source Software Development, January draft, <http://opensource.mit.edu/papers/healyschussman.pdf>
- Koch, Stefan. 2005. "Evolution of {O}pen {S}ource {S}oftware Systems - A Large-Scale Investigation", *Proceedings of the 1st International Conference on Open Source Systems*, Genova, Italy, July.
- Krishnamurthy, Sandeep. 2002. Cave or Community? An empirical examination of 100 mature open source projects, *First Monday* vol 7, no. 6 (June), http://www.firstmonday.org/issues/issue7_6/krishnamurthy/
- Lehman, Manny M., Ramil, Juan F., Wernick, P. D., Perry, D., E., and Turski, W. M. 1997. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, (November), p. 20.
- Mockus, Audris, Fielding, Roy T. and Herbsleb, James D. 2002. "Two case studies of Open Source Software development: Apache and Mozilla", *ACM Transactions on Software Engineering and Methodology*, volume 11, number 3, pages 309-346,
- Robles, Gregorio. 2006. *Empirical software engineering research on libre software: data sources, methodologies and results*. Doctoral thesis. Universidad Rey Juan Carlos, Madrid, February. Available online at <http://libresoft.urjc.es/grex/phd>
- Robles, Gregorio et al. 2001. WIDI: Who Is Doing It?, <http://widi.berlios.de/paper/study.html>
- Robles, Gregorio, Amor, Juan José, González-Barahona, Jesús M. and Herraiz, Israel. 2005. "Evolution and Growth in Large Libre Software Projects", *Proceedings of the International Workshop on Principles in Software Evolution*, Lisbon, Portugal, September, 165 - 174;
- Schach, Stephen R. and Offutt, A. Jefferson. 2002, "On the Nonmaintainability of Open-Source

Ghosh and David *Relating social structure to technical structure: findings from the Linux kernel*

Software," *Proceedings of the 2nd Workshop on Open Source Software Engineering*, Orlando, FL, May 2002.

Succi, Giancarlo, Paulson, J. W. and Eberlein, A. 2001. "Preliminary Results from an Empirical Study on the Growth of Open Source and Commercial Software Products", EDSER-3 Workshop (co-located with *International Conference on Software Engineering ICSE 2001*), May, Toronto, Canada

Trung T. Dinh-Trong and Bieman, James M. 2005. "The FreeBSD Project: A Replication Case Study of Open Source Development", *IEEE Transactions on Software Engineering*, volume 31, number 6, pages 481-494, June

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>

Wendel de Joode, Ruben van and Kemp, Jeroen. 2001. "The Strategy Finding Task Within Collaborative Networks, Based on an Exemplary Case of the Linux Community", <http://opensource.mit.edu/papers/dejoode.pdf>;

Wheeler, David, *More Than a Gigabuck: Estimating GNU/Linux's Size* , July 2001, <http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html>