Noname manuscript No.
(will be inserted by the editor)

Effort Estimation for FLOSS Projects: A Study of the Linux Kernel

Andrea Capiluppi · Daniel Izquierdo-Cortázar

Received: date / Accepted: date

Abstract The differentiation in "core" and "peripheral" roles has been established and largely accepted within the Free/Libre/Open Source Software (FLOSS) development approach, assigning to each role different responsibilities and productivity patterns. A further, cross-cutting characterization of developers within the FLOSS approach could be formulated clustering developers into "time slots", and different patterns of activity and effort assigned to such slots. Such analysis, if replicated, could be used not only to compare different FLOSS communities, or to evaluate their stability and maturity, but also to determine how the effort is distributed in a given period, and to estimate future needs in proximity of key points (e.g., major releases).

This study analyses the activity patterns within the Linux kernel project, at first focusing on the overall distribution of effort and activity within weeks and days; then, dividing each day into three 8-hour time slots, and around major releases. Such analyses have the objective to evaluate effort, productivity and types of activity globally and around major releases, compare these patterns with traditional software products and processes, in turn identifying company-driven projects (i.e., working mainly during office hours) among FLOSS endeavors.

The results of this research show that, overall, the effort within the Linux kernel community is constant (albeit at different levels) throughout the week, signalling the need of updated estimation models, different from those used in traditional 9am-5pm, Monday to Friday commercial companies. It becomes also evident that the activity *before* a release is vastly different from *after* a release, and that the code quality decreases in specific time slots (notably in late night hours), which later will require additional maintenance efforts.

Dr A. Capiluppi
University of East London, UK
E-mail: a.capiluppi@uel.ac.uk

D. Izquierdo-Cortázar
Universidad Rey Juan Carlos, Spain
E-mail: dizquierdo@gsyc.es

1 Introduction

Software development productivity measurement and cost estimation has been a research topic for more than 3 decades [1], [2], [3]. The vast majority of empirical studies has so far involved data from proprietary software projects [4]: albeit an increasing number of governments, non-governmental organizations and companies seem interested in using, evaluating and contributing to FLOSS, effort estimation models or other measurement-based models are not in general used within FLOSS communities [4]. Indeed, such exploration and quantification of productivity, and how a FLOSS community manages and allocates effort around a major release, may help in comparing FLOSS projects both with proprietary systems, and between large FLOSS communities. Furthermore, such productivity modeling can also help to identify a baseline to measure the possible impact of changes in, for example, processes, methods and tools used by FLOSS communities.

What has been generally accepted when dealing with FLOSS productivity is that there is an increase in productivity as long as FLOSS developers progress in their statuses within a project, along the clusters depicted in the so called "onion model" [5], [6]: the external layer of this representation consists of *users*, strictly speaking not representing developers, but nonetheless forming a valuable community for both the diffusion of a FLOSS product, and the testing of their functionalities. The *contributors* represent the next layer, producing source code and fixes, apart from providing feedback and discussion; this layer is known as being more numerous than the one with users. Finally, *core developers*, representing the centre of the onion, provide most of the work needed both in the creation, and in the maintenance, of new or existing content, and their productivity is an order of magnitude higher than the contributors. It has been also argued that the core team must be small [7], in order to keep a tight control over the core system. It has also been found that the coordination issues of traditional software systems (e.g., the Brooks' law [8]) still apply within FLOSS core teams, while such issues are rapidly decreasing in relevance when considering the other layers of such model [9].

Based on such clustering, the objective of this research is to develop a framework for FLOSS effort estimation, by grouping developers around different *time slots*, and by considering "days of the week" or "hours in a day" as cross-cutting attributes for effort and productivity models. The rationale for doing so derives from both a lack of such differentiation in the current literature, and the results obtained in a previous work [10], when analysing the effort produced by a UK Agile company. Among the other results, it was found that the pattern of activity (in terms of its commits towards the Software Configuration Management repository, SCM) could be described by a traditional 9am-to-5pm commitment, and a propensity to leave some days of the week where the coding activities are less sustained (typically, Fridays). The two graphs in Figure 1 display the derived hourly and the weekly activity patterns in this company.

Performing a similar analysis for FLOSS projects could help in better understanding how the FLOSS development works, and whether it departs from using "traditional" effort estimation models. On the one hand, it could highlight productivity patterns around specific dates (e.g., when a major release is made public). On

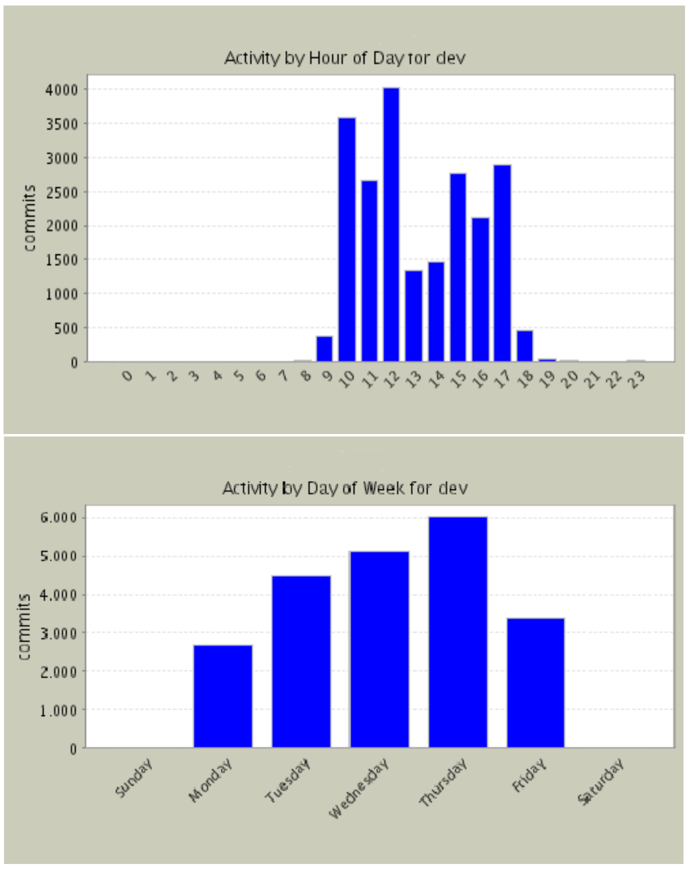

Fig. 1 Aggregated commits divided by hour of the day)

the other hand, it could determine whether specific time slots result more productive, or are more prone to low quality contributions, than others.

In both cases, the wealth of data coming from FLOSS projects could help producing and replicating ad-hoc estimation models, eventually differentiating *company-driven* projects from *community-driven* endeavors. FLOSS projects backed from large company organizations (i.e., company-driven) should reflect developers with a more traditional, 9am-5pm activity patterns, commit policies and so forth. The *community-driven* FLOSS projects should instead follow more continuous working patterns, since developers are working in their spare time, and outside normal “office hours”. If identified and confirmed, such emerging patterns would present new, specific issues: first, it would emerge the need of differentiating the effort estimation models based on the periods of activity, by means of weights and triggers of model-switching. Second, it would emerge the need of monitoring tools in specific time intervals, or parts of the day, in order to properly monitor the diverse productivity at certain times of the day, or in specific days of the week.

So far, this research has achieved three main contributions:

1. the first contribution is to demonstrate that the patterns of work within the selected case study (the Linux kernel) are different from those found in a traditional software development team (even in the case of using an advanced process, as the Agile methodology).
2. The second contribution is to present the outcomes when studying the development of the Linux kernel along specific periods of the day (e.g., *time slots*), and in specific periods (around major releases), with the aim of investigating the changes in productivity and code quality during such periods.
3. Finally, the approach used in this paper has specifically considered a “Git” SCM repository, which offers additional information, not covered in other such repositories, and not used in previous studies on FLOSS systems. Differently from other configuration management systems (such as CVS or SVN), a Git repository retains the information about both the authors and their local submission dates, rather than aggregating the latter into the central server’s time [11]. With this information, it is possible to group the developers effort in the effective time of the day when such actions were performed, which provides a valuable information when a distributed, trans-national development approach is considered (as the FLOSS model requires).

2 Vocabulary and Study Planning

This section introduces the definitions used in the following empirical study and presents the general objective of this work, and it does that in the formal way proposed by the *Goal-Question-Metric* (GQM) framework [12]. The GQM approach evaluates whether a goal has been reached, by associating that goal with questions that explain it from an operational point of view, and providing the basis for applying metrics to answer these questions. This study follows this approach by developing, from the wider goal of this research, the necessary questions to address the goal and then determining the metrics necessary for answering the questions.

Goal: the long term goal of this research is to define, validate and update productivity models for FLOSS projects, and to differentiate them from existing proprietary models.

Question: In this paper, and considering the Linux Kernel as a case study, the following research questions have been evaluated:

1. Do Linux developers work specifically during some days of the week, or some hours of the day?
2. Is there a statistically significant difference in the activity during various parts of the day?
3. Is there any part of the day that is more prone to issues of code quality?
4. Is there a statistically significant difference in the activity before and after a major release in the Linux kernel?
5. Are different estimation models needed for taking into account the activity in the various timezones of the day?

Metrics: Two empirical studies have been carried out in this paper, one related to the characterization of the overall activity of commits by committers during the whole development log of the Linux kernel; the other focused only on the major releases between (and including) 2.6.12 and 2.6.34, and analysing the development activity both one week before, and one week after a major release. The CVSAnalY tool [13] was used in order to retrieve information from the log found in all the source code management systems. Specifically, we are interested in the table *scmlog* where most of the log information is stored. This table contains some useful fields such as *date, author_id and committer_id* which help to calculate the data we need. In order to test the various hypotheses, *two tail*, *heteroscedastic* t-test will be used to compare pairs of samples, testing whether the observations on one sample are not in any way related to the observations in the other.

2.1 Definitions and Empirical Approach

The definitions of this study are:

– *Commit (or revision)*: change on the source code submitted to the source code management system. This updates the current version of the tree directory with a new set of changes. Those changes are generally summarized in a *patch* which is a set of lines with specific information about the affected files, but also about the affected lines.
– *Committer*: this is the person who has rights to commit a change into the source code.
– *Major release*: this paper will focus on specific points when higher activity is detected, namely the releases of the Linux were made publicly available. The releases studied in this paper are the ones contained (or migrated) within the Git repository during the 2.6 branch of development, starting from release 2.6.12 and including release 2.6.34. In total, an overall of 23 releases was analysed, spanning some 5 years of development under the Git repository.
– *Author*: A commit could be committed by a given committer, but she may not be the real author. Some SCMs offers this information, and the Git SCM provides a specific field for this.
– *Timezones*: in this paper any day is divided in three 8-hours sections, and we define “office hours” (OH) the period from 9:00 to 17:00 between Mondays and Fridays; we define “after office” (AO) the period from 17:00 to 1:00, while “late night” (LN) from 1:00 to 9:00.
– *Complexity*: since the Linux kernel is developed mainly using the C programming language, the definition of complexity used in this paper is taken from the McCabe cyclomatic index [14], [15].

2.2 Empirical Approach – Overall Activity

The first part of the paper is devoted to the characterization of the development activity within the Linux kernel: the following empirical approach was followed in this first part:

1. **Git clone:** at first, the Linux Git repository[1] was cloned and stored locally. As reported above, this repository spans the late life-cycle of the Linux Kernel (since April 2005), when the project was moved to the Git repository.
2. **Data pre-parsing:** the information contained in the log of such repository was parsed into commonly used results: the CVSAnalY toolset was used for this purpose, saving each commit ID, and the relevant data along that commit, including the time, the authors and the committer, and the rational of such commits.
3. **Time and full-path parsing:** further to the pre-parsing by the CVSAnalY tool, the *time* attribute of each commit was clustered in one of three slots, "office hours", "after office" or "late hours", depending on the hour of such commit.
4. **Major release dates:** from the overall activity log of the Linux kernel (obtained by issuing the "git log" command), the dates of each of the aforementioned releases was clearly identified by a "release announcement" statement, and cross-validated, for each release, with the upload date to re-distribution websites (e.g., `http://www.kernel.org/pub/linux/kernel/v2.6/`).
5. **Identify commits before and after a release:** in order to identify the list of commits performed during the seven days before a major release (but excluding the actual day of release), the database produced by CVSAnalY was queried starting from the midnight of the first day, till the 23:59 of the seventh day[2]
6. **Added, Deleted and Modified lines:** each commit is parsed with the 'diffstat' utility, which uses the more common 'diff' program to define summaries of added, deleted and modified lines within a large, complex set of changes. In particular, for each commit, the switch "-m" is used to summarize a large chunk of modifications in a readable format.

2.3 Empirical Approach – Complexity

The second part of this research is devoted to studying whether one of the time zones used in this paper is more prone to unprofessional code than other parts of the day. Differently from the first part of the paper, this second analysis has not produced an overall view of how the complexity is characterized in the whole life-cycle, but it only focuses on the seven days before and the seven after a major release, as defined above.

The following steps were followed to determine how the complexity was introduced, increased or reduced along various commits or revisions:

1. **Identify files affected in a commit:** based on the list of commits executed either before (pre-) or after (post-) a major release, a Git repository gives the opportunity to display all such changes through the "git show" command. The output of such command to display a summary of files affected by a revision (say, 'c'), as in "`git show c | diffstat -m`". As a cross-validation of such results, we used the information stored by CVSAnalY in the table "actions".

[1] As found in `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git`

[2] In a SQL statement, *where date* $\geq$ *'2005-06-09 00:00:00' and date* $\leq$ *'2005-06-24 23:59:59'*

2. **Extracting the full path of files:** Since the basic CVSAnalY only extracts file names, the full path of the files affected in a specific commit was extracted, in order to properly track moved and renamed files. The command issued for extracting the full paths of the files affected in a commit 'c', is "`git show c | diffstat -p1 -w70`".
3. **Evaluating the previous revision of a file:** any file in the Git repository, after being added, will go through a series of revisions, ordered by the date when each was performed. If, say, the three files A, B and C were modified in revision *rev(t)* (Figure 2), each will have a previous revision where they were modified or firstly added (in the example, B in *rev(t-1)*, C in *rev(t-2)* and A in *rev(t-3)*). Given a revision 'r' of the file 'f', the Git repository will show how the file 'f' was composed in that specific revision 'r', by issuing the command `git show r:f`. In this way, it is possible to compare two revisions of the same file, and to check whether the changes inputed by a developer affected its structure.
4. **Evaluation of the change in complexity:** having the two subsequent revisions of the same file, it is possible to evaluate both the complexity of its functions (since the vast majority of the Linux kernel is implemented in the C programming language), and the overall complexity of the same file, in the two subsequent revisions. By cross-cutting this analysis with the information on the time of each revision, it is possible to conclude whether in any of the time slots developers added or removed complexity, or whether the change left the same complexity unmodified.

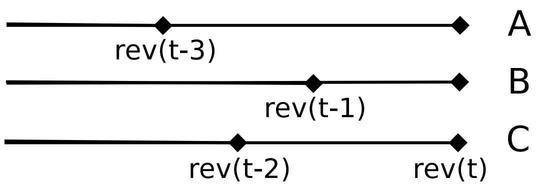

Fig. 2 Evaluating previous revisions of files

3 Results – Development Activity

As mentioned above, the case study is the Linux Kernel which has been previously studied several times and from several points of view ([16], [17], [18], [19]). Two aspects are presented below: the first considers the whole evolution log of the Linux Kernel (since April 2005, when the overall data has been moved to the Git repository) and it displays the patterns of activity in terms of week-days and hours worked on by the Linux developers (irrespective of them being "core" or "peripheral" developers). The second focuses on specific weeks of the Linux kernel development, justifying this choice with the observed bias in the distribution of effort, and attributed to the presence of major releases.

3.1 Results – Weekly and Hourly Activity

In order to compare and contrast the findings of the activity patterns during working hours and throughout a week of traditionally developed software (Figure 1), the following section presents the analysis of the Linux kernel development under a similar perspective.

Figure 3 (left) shows the analysis of the overall activity within the Linux Kernel during the day, as recorded within the Git log. The first observation is that the work/no-work distinction, found within the commercial counterpart [10] (and depicted in Figure 1 left), is not easily applicable to the Linux kernel development. The activity performed between 9am and 5pm (corresponding to the "office hours") accounts for some 55% of the overall amount of commits; some 31% of the overall activity is produced during the "after work" interval, or between 5pm and 1am; finally, some 14% of the activity is performed during the "late hours", or between 1am and 9am. The second and third slots of activity therefore represent a consistent departure from the commercial counterpart studied in [10], reflecting a traditional pattern of activity since most of the commits appear during the "office hours" (Figure 3, left). On the contrary, in the Linux kernel, the most active timezone is found between 2pm and 4pm. Specifically at 3 pm we can see a peak of activity which gradually decreases during the after-office hours.

Figure 3 (right) shows a complementary picture. The blending between a company-driven community (which tends to work in *office time*), and a community-driven project (where developers tend to work mostly on their spare time) is evident in the distribution of activity throughout the week. In this figure, we divide the week in the weekdays and calculate the aggregated number of commits for the whole life of the project. This figure shows how people in the Linux Kernel tend to work during the weekdays: the first, clearly defined period is the interval "Monday - Friday", where the number of commits is daily more than 30,000. The second period of activity appears specifically during the Saturdays and Sundays, where the number of commits jointly reaches some 30,000 commits (i.e., the same amount of commits achieved in any other day of the week). In summary, the comparison with a traditional commercial system shows that the Linux Kernel benefits overall from one "extra" day of development per week (6 days with similar productivity out of 7), whereas the observed Agile system benefits from 5 (unequally productive) days per week (Figure 1 right).

The observed patterns, in the Linux kernel and the Agile company, pose an issue of how to quantitatively describe the observed effort, and how to formulate an effort estimation model. Since the distribution of effort is predefined throughout the office hours in a commercial environment, the effort is only applied in that slot: therefore, when expressing the effort as a function of the performed activity (e.g., amount of commits, lines added, modified or deleted; files added, modified or deleted; etc.) the modeled Agile commercial system would need to be modeled by an equation such as

$$E(t) = f(activity(t)_{OH}) \tag{1}$$

where $E_{Commercial}(t)$ is the effort by developers during the period t (daily, weekly, monthly, etc), while $f(Commits(t)_{OH}$ is a function of the amount of com-

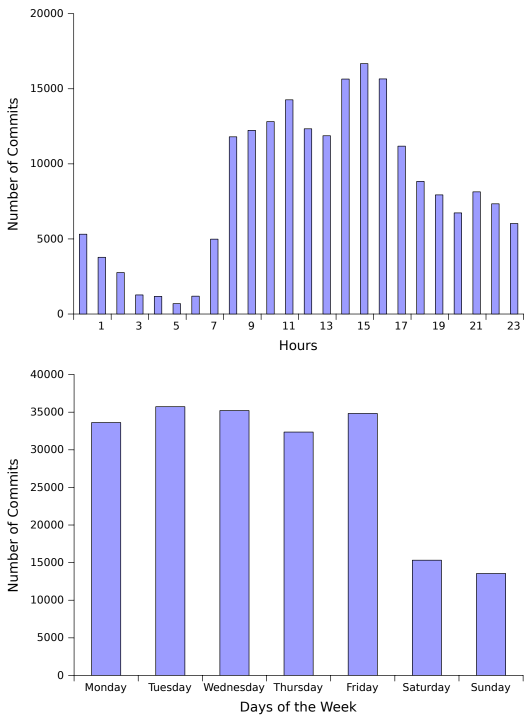

Fig. 3 Aggregated commits divided by hour of the day (left), and by the activity during the week (right)

mits, during the same period, but only within the office hours boundaries (i.e., 9am to 5pm).

On the other hand, when modeling the overall activity seen in the Linux kernel (and most likely other FLOSS systems), and taking into account the three timezones (Office Hours, OH; After Office, AO; Late Hours, LH), one should also take into account the other timezones, and weigh them appropriately:

$$E(t) = w_{OH}*f(activity_{OH}(t))+w_{AO}*f(activity_{AO}(t))+w_{LN}*f(activity_{LH}(t)) \quad (2)$$

where w_{OH} is the weight given to the activity observed within the Office Hour slot; w_{AO} the weight to the After Office slot; and w_{LN} the weight to the Late Night slot. In the case of the reported Linux kernel, the overall activity observed in this project produce the following weights: $w_{OH} = 0.55$, $w_{AO} = 0.31$ and $w_{LN} = 0.14$.

3.2 Results – Types of Activity

The overall activity shown above has the advantage of proposing the global picture of the development within the Linux kernel, without revealing whether some parts of the day were more prone to additions, deletions or modifications. In order to perform a more focused analysis of the *type* of activity occurring in the various parts of the day, a number of "random" weeks were selected to analyse whether the division of a day in three parts can shed further insights on how work is performed within the Linux kernel.

The analysis reported below refers to the week between 13/04/2009 (Monday) and 19/04/2009 (Sunday), where all the 838 performed commits have been analysed for the purpose. Figure 4 reports how the changes evolve during such week. These changes are divided in six different groups: the three main groups are given by the three defined timezones and for each of them, we have calculated the number of added and removed lines. In general, this distribution of the work follows the initial distribution shown in the previous figures, except for the Wednesday. This seems to be an outlier that does not follow the general tendency in amount of work.

For the mentioned figure, we can observe how the number of lines handled during the weekend (even when we select the whole day and not divided by timezones) is really low, being developed the main activity in this specific week during the week days.

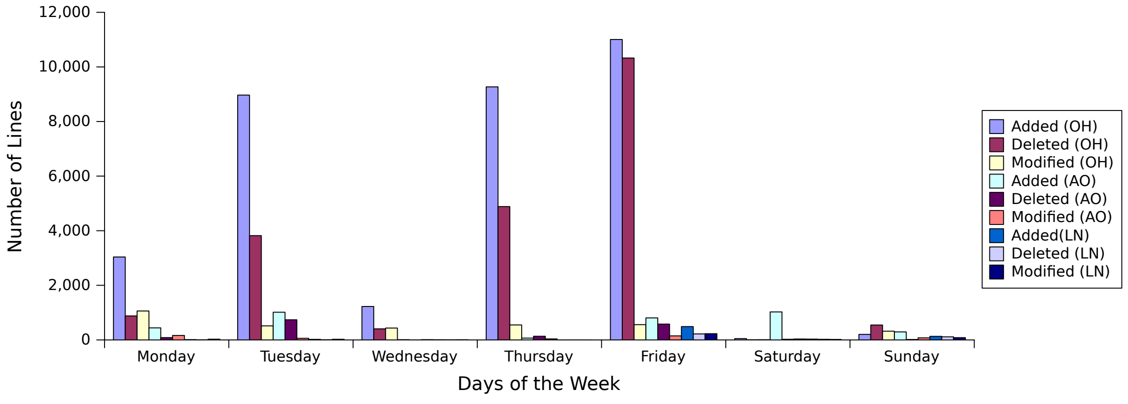

Fig. 4 Size of Changes for the week 13th-19th of April, 2009)

The figure 5 shows a bit more of information for each day. In the specific case of the week days, we observe how the quantity of lines (added or removed) in a given day is really high compared to the rest of the day, reaching some days almost the 100% of the total modifications. On the other hand, we can see how in the weekends, the activity developed by the people (even out of the office time [3]) is really low, but developed out of the office time. In this case, the activity developed during the weekend reaches up to an 80% on Saturday, and a 40% on Sunday.

Table 1 finally displays, for the aforementioned week, the changes observed, and divides them in three categories: added, deleted and changed lines. As also observed

[3] We provide the results for the office time during the weekends just to observe if there is a continuous activity during the mornings. However, it has not happened since most of the activity, for instance, during the Saturday, is developed during the afternoon and in the following.

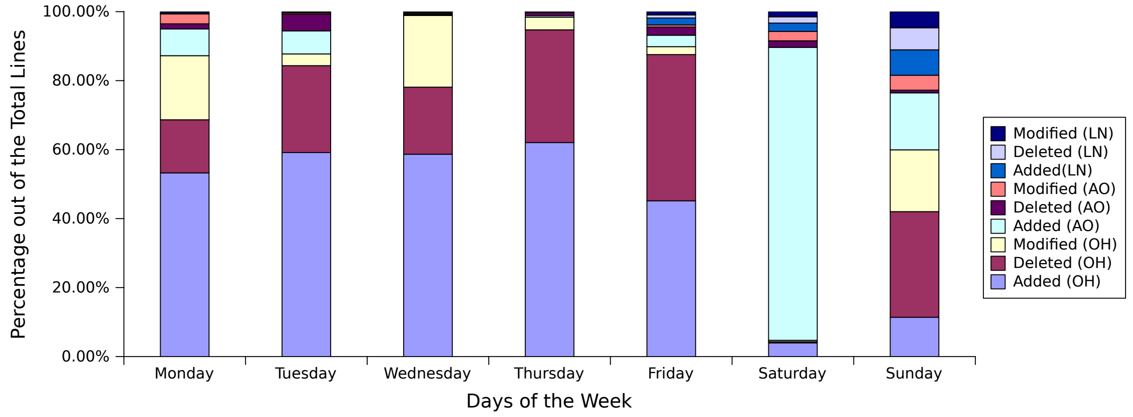

Fig. 5 Percentage of Changes for the week 13th-19th of April, 2009)

in Figures 4 and 5, most of the activity is achieved during the day, in the timezone 9am-5pm, with an overall count of 555 commits (some 66% in the week). What is interesting to note is that, albeit a lower activity is noticeable during the late hours, the average number of added and deleted lines is vastly different from the other two periods of the day, the average number of added lines being more than double than the rest of the day, and the average number of deleted lines being more than four-some with respect to the rest of the day.

Hours	$9am - 5pm$	$5pm - 1am$	$1am - 9am$
Nr of commits	555	187	96
Avg nr of added lines	41	51.5	117
Avg nr of deleted lines	20	26	112
Avg nr of changed lines	1.6	0.6	0.47

Table 1 Average size of changes, differentiated by timezones and type of change

The initial results were tested and compared with other randomly selected weeks, but the findings reported above were not thoroughly confirmed in the other sampled weeks. Investigating further, it was found that the sequence of major and minor releases within the development plays a distorting role in applying effort by committers towards a specific deadline. Figure 6 shows how the amount of commits vary when considering seven days before and seven days after the “peak” of activity represented by the actual day when the 2.6.14 release was made public. Therefore it was decided that a study for characterizing the types of activity observed in the Linux kernel should take into account such sequence of releases: the next section details and analyses the activity observed seven days before and seven days after the date of a major release (while excluding the peak of the same day), for the purpose of producing estimation models based on the types of actions observed in the development.

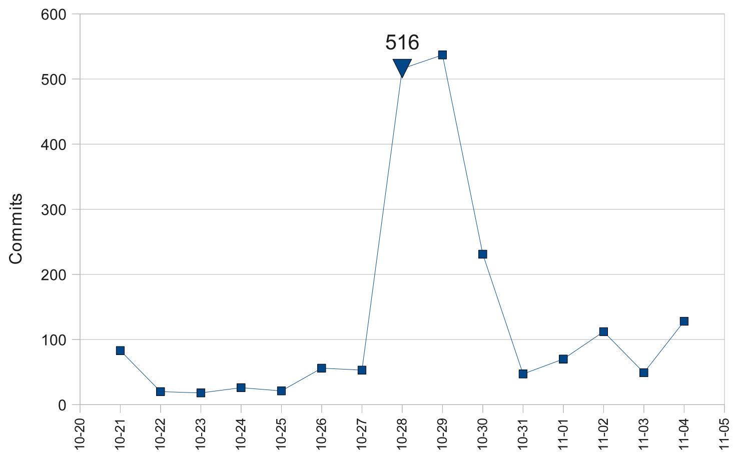

Fig. 6 Activity one week before and one week after the 2.6.14 release

3.3 Results – Before and After a Major Release

When considering the available development history of the Linux kernel hosted within the Git repository, 23 major releases were studied in this section, from 2.6.12 to 2.6.34. Each was analysed with respect to the amount of *commits*; *authors* and *committers*; *added*, *deleted* and *modified* lines as recorded both seven days before, and seven days after the date of each public release.

The results of such analysis are reported, as longitudinal trends in the amount of commits per release in Figure 7, and in the tabular form of Table 2, detailing for each studied characteristic, its mean and variance value, both a week before and a week after a major release.

The following findings have been observed:

– Throughout all the 23 studied releases, the average amount of commits-per-release is somewhat similar during the OH and AO slots, and both pre- and post- major releases;
– The average amount of commits-per-release during the slot LN clearly lower than the OH and AO, both pre- and post- major releases, signaling a lower activity in such slot;
– The similarity between the OH and AO slots is consistent for all the studied metrics (authors; added, deleted and modified lines). The LN slot instead consistently presents a lower level of activity;
– Despite the lower amount of activity, the Linux kernel had an increasing number of people working during the LN slot, in both the pre- and post-week periods. The pre-2.6.12 week only had 2 authors active during the LN slot, while the pre-2.6.34 week had some 311 authors in charge of commits; the post-2.6.12 week benefited from 84 authors, and the post-2.6.34 week from 640 authors. Figure 8 describes the intersections, for all the releases of the set of authors and committers working on the OH, AO and LN slots.

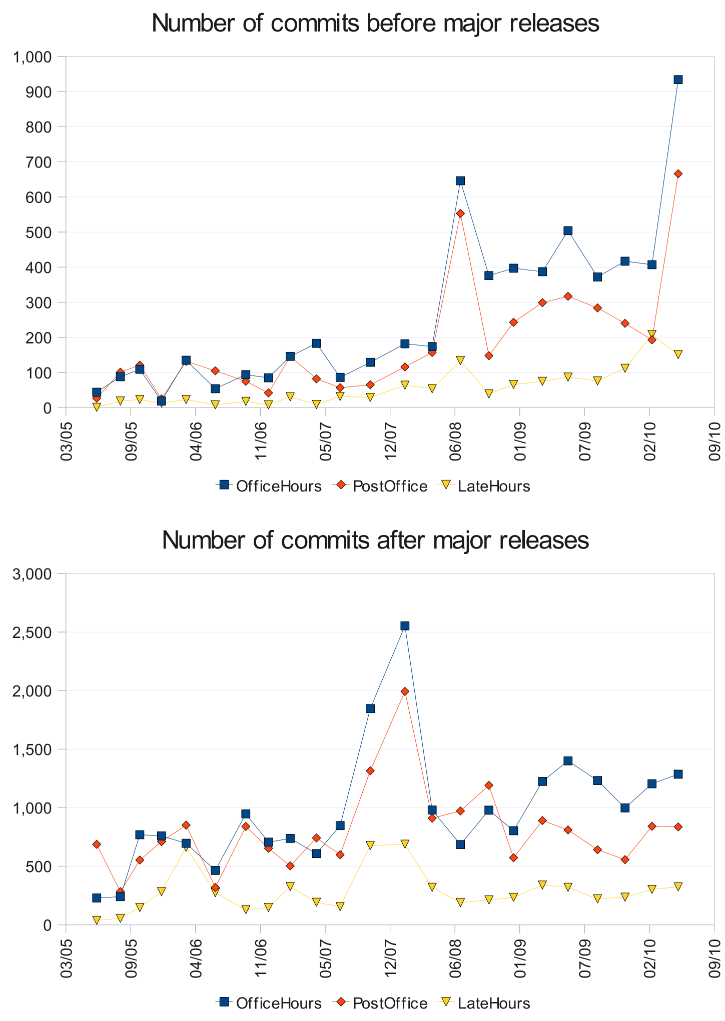

Fig. 7 Aggregated commits divided by hour of the day)

– The distributions of all the measured characteristics were found to be statistically different, when considering the pre- and post-weeks: for example, the distribution of commits in the OH slot before releases (51, 94, 121, 24, 141, 74, 103, 88, 152, 191, 94, 149, 196, 179, 682, 399, 435, 417, 530, 403, 462, 425, 959) is statistically different from the distribution of commits after releases (258, 269, 824, 797, 739, 484, 963, 722, 766, 631, 884, 1891, 2571, 1018, 739, 1062, 845, 1287, 1498, 1288, 1048, 1272, 1361) when applying the t-test.

Based on such findings, the effort estimation equation in (2), and the term *activity(t)* should be tailored to reflect such differentiation in both the time slots, and depending on whether the activity is monitored and estimated in the weeks before or

	Attribute	Mean (pre-)	Mean (post-)	Variance (pre-)	Variance (post-)	$t-test$ (pre- *vs* post-)
Office Hours	Commits	271	964	50,954.8	264,659	8.73e-07
	Authors	541	1,954	1.89e+05	1.34e+06	3.77e-06
	Added lines	17,715	98,608	3.29e+08	6.86e+09	6e-05
	Deleted lines	8,845	44,856	9.08e+07	3.37e+09	0.00368
	Modified lines	4,704	14,857	2.73e+07	9.25e+07	4.4e-05
After Office	Commits	200	786	2.53E+004	1.33E+005	3.57E-08
	Authors	391	1,621	9.41E+004	5.90E+005	3.88E-08
	Added lines	10,621	65,393	1.30E+008	2.19E+009	6.03E-06
	Deleted lines	6,931	36,519	1.33E+008	1.35E+009	0.00052
	Modified lines	2,822	13,147	1.09E+007	5.63E+007	6.04E-07
Late Hours	Commits	59	295	2.90E+003	4.08E+004	6.25E-006
	Authors	122	699	1.41E+004	3.71E+005	8.30E-005
	Added lines	3,433	18,500	2.10E+007	2.42E+008	7.22E-005
	Deleted lines	1408	9,936	5.46E+006	7.84E+007	7.41E-005
	Modified lines	704	4,766	5.92E+005	1.57E+007	3.36E-005

Table 2 Activity one week before and one week after major releases, clustered by time-slots

after a major release. A list of equations for the activity could be obtained as follows, and based on the assumption that the actions of “adding”, “deleting” and “modifying” lines (or files) are exhaustive of the type of actions perfomed by developers during the period t (say, hourly, daily, weekly, etc):

$$activity_j^i(t) = w_j^i * f(Add_j^i(t), Del_j^i(t), Mod_j^i(t)) \tag{3}$$

where i the index indicates whether the activity is observed either “before” or “after” a release; the j index instead can be used to differentiate between the activity as seen in the OH, AO and LN slots. The w_j^i terms then become the weights of the actions performed in a specific week and during a given time slot.

4 Results – Complexity in Time Slots

The second part of this research has been focused on the presence of complexity (measured by the McCabe cyclomatic index), and its changes, within the source files of the Linux kernel. As reported above, this second study was performed focusing on the activity:

– of the 23 releases found between April 2005 and June 2010, and
– differentiating the results in “one week before” a release from those “one week after”, and finally
– clustering each day of activity in the three time-zones: OH, AO and LN.

The analysis was performed only on the “.c” source files and “.h” headers[4] that underwent changes during the pre- and post-release weeks. For each of the commits

[4] This was done to properly evaluate the McCabe cyclomatic complexity for source files developed in the C “procedural” language

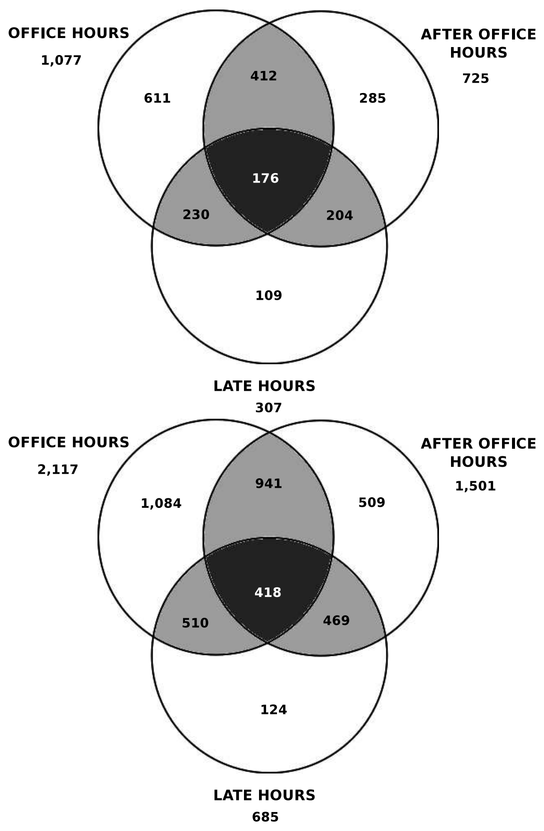

Fig. 8 Intersections, all releases, committers (left) and authors (right)

performed in such weeks, it was studied whether the changes pushed by committers did alter the overall complexity of the affected files. Only the “modified” files were considered in such evaluation, therefore leaving aside the addition of new files (which adds new source code, let aside new complexity). To the best of our knowledge, this is the first time that an analysis of how single source files changed within subsequent commits is performed in a large case study.

The results are reported in Table 3: they are clustered around the three time slots (OH, AO and LN) and summarized in relative terms. Each time slot presents two series of data, the first (2nd, 4th and 6th columns) depicting the amount of files which underwent an increase of complexity, the second series (3rd, 5th and 7th columns) the amount of files which had a decrease of their overall McCabe cyclomatic number instead: both series are relative numbers, and divided by the amount of files handled in the same week. The following observations were made:

1. During the pre-release weeks, the activity during late night hours has been, so far, the most likely to increase the complexity when modifying the source files. In other words, in 16 releases out of 23 (70%), during the LN slot the committers and authors have been responsible of changes that have increased the files’ complexity

more often than in the Office Hour time slots. This is a somewhat indirect proof that some time slots are more prone to "unprofessional" maintenance to existing source files than other slots (say, the Office Hours) This is also shown in the distribution of such ratios in Figure 9.

2. On the contrary, during the *post-release* weeks, the Office hour slot initially has consistently seeded more complexity into the source files. In more recent releases, instead both the After Office and the Late Night slots have started to insert more complexity into files, as compared to the office Hour slot, signaling again the importance of such slots in seeding more complexity within modified files.
3. The distributions of source files undergoing increases of complexity is statistically different in each time slot, when performing a t-test comparison: for instance, the global amount of files undergoing increases of complexity in the OH slot presents statistically relevant differences when comparing the week before[5] and the week after[6] a release, when applying the two-tail t-test (1.174E-007).
4. The patterns in the *decrease* of complexity show instead a different perspective: during the weeks before releases, no major differentiation between the various time slots is visible, each presenting a fluctuating and inconsistent behavior. On the other hand, in the weeks after the releases the slots are devoted to either overall increases of complexity, or decreases, but not both (as in the pre-weeks).

Considering the relation for effort estimation in Equation (2), it is possible to discriminate, within the "activity" term, the portion of such activity devoted to the increase of complexity, the portion that increases the complexity, and the portion that does not affect the complexity. Each of the terms $activity_{OH}(t)$, $activity_{AO}(t)$ and $activity_{LH}(t)$ can be further expanded in the following:

$$activity_{OH}(t) = w_{OH}^{IC} * aIC_{OH}(t) + w_{OH}^{DC} * aDC_{OH}(t) + w_{OH}^{WChC} * aWChC_{OH}(t) \tag{4}$$

$$activity_{AO}(t) = w_{AO}^{IC} * aIC_{AO}(t) + w_{AO}^{DC} * aDC_{AO}(t)) + w_{AO}^{WChC} * aWChC_{AO}(t) \tag{5}$$

$$activity_{LN}(t) = w_{LN}^{IC} * aIC_{LN}(t) + w_{LN}^{DC} * aDC_{LN}(t) + w_{LN}^{WChC} * aWChC_{LN}(t) \tag{6}$$

where w_i^{IC}, w_i^{DC} and w_i^{WChC} are the weights of the activities for increasing (IC), decreasing (DC) or without changes (WChC) in the complexity of the source files during time slot i. The terms $aIC_{LN}(t)$, $aIC_{LN}(t)$ and $aIC_{LN}(t)$ represent instead the actual activities of increasing, reducing or not affecting the overall complexity of files at time t, respectively.

[5] Number of source files where complexity increases, during the week **before** a release: 13, 33, 28, 1, 48, 33, 30, 48, 54, 87, 36, 37, 72, 56, 190, 149, 129, 128, 237, 216, 177, 146, 314

[6] Number of source files where complexity increases, during the week **after** a release: 100, 102, 256, 343, 245, 172, 409, 255, 254, 273, 346, 712, 771, 360, 271, 349, 324, 428, 523, 349, 471, 399, 493

	OH		AO		LN	
	INCR	DECR	**INCR**	DECR	**INCR**	DECR
2.6.12-pre	**0.24**	0.15	0.11	0.30	0.00	0.00
2.6.13-pre	0.21	0.13	0.16	0.05	**0.23**	0.12
2.6.14-pre	0.16	0.22	**0.22**	0.09	**0.18**	0.21
2.6.15-pre	0.07	0.20	**0.21**	0.12	**0.40**	0.13
2.6.16-pre	**0.28**	0.11	0.16	0.08	0.13	0.13
2.6.17-pre	0.21	0.16	0.17	0.03	**0.38**	0.00
2.6.18-pre	0.10	0.03	0.16	0.13	**0.20**	0.16
2.6.19-pre	0.13	0.12	**0.17**	0.03	**0.22**	0.22
2.6.20-pre	0.26	0.13	0.15	0.15	**0.42**	0.16
2.6.21-pre	**0.38**	0.13	0.28	0.10	0.13	0.04
2.6.22-pre	**0.39**	0.07	0.08	0.10	0.15	0.07
2.6.23-pre	0.27	0.14	**0.37**	0.19	**0.38**	0.15
2.6.24-pre	0.26	0.14	**0.27**	0.08	**0.27**	0.20
2.6.25-pre	0.17	0.08	**0.18**	0.10	**0.22**	0.10
2.6.26-pre	0.28	0.13	0.26	0.11	**0.40**	0.07
2.6.27-pre	0.17	0.09	**0.26**	0.07	**0.24**	0.08
2.6.28-pre	0.28	0.11	0.18	0.12	**0.29**	0.06
2.6.29-pre	0.30	0.13	0.21	0.13	**0.38**	0.14
2.6.30-pre	**0.33**	0.13	0.30	0.15	0.16	0.05
2.6.31-pre	**0.39**	0.16	0.34	0.08	0.15	0.14
2.6.32-pre	0.25	0.11	0.19	0.08	**0.29**	0.10
2.6.33-pre	0.25	0.19	0.22	0.23	**0.27**	0.21
2.6.34-pre	**0.24**	0.11	0.15	0.09	0.12	0.06
	INCR	DECR	**INCR**	DECR	**INCR**	DECR
2.6.12-post	**0.25**	0.13	0.24	0.13	0.05	0.04
2.6.13-post	**0.25**	0.06	0.16	0.13	0.22	0.05
2.6.14-post	0.16	0.14	0.13	0.09	**0.27**	0.05
2.6.15-post	0.17	0.08	**0.18**	0.26	**0.20**	0.14
2.6.16-post	0.21	0.12	**0.24**	0.17	0.16	0.14
2.6.17-post	0.24	0.09	**0.25**	0.14	0.17	0.13
2.6.18-post	0.28	0.14	**0.29**	0.12	0.22	0.12
2.6.19-post	**0.22**	0.14	0.17	0.14	0.10	0.18
2.6.20-post	**0.20**	0.12	0.14	0.09	0.13	0.15
2.6.21-post	**0.26**	0.14	0.24	0.18	0.16	0.13
2.6.22-post	**0.24**	0.12	0.20	0.12	0.23	0.13
2.6.23-post	**0.21**	0.15	0.16	0.22	0.16	0.20
2.6.24-post	**0.27**	0.13	0.19	0.13	0.24	0.09
2.6.25-post	**0.25**	0.12	0.19	0.10	0.18	0.12
2.6.26-post	**0.25**	0.15	0.20	0.12	0.20	0.22
2.6.27-post	**0.27**	0.15	0.21	0.15	0.25	0.06
2.6.28-post	**0.29**	0.13	0.22	0.15	0.18	0.10
2.6.29-post	0.22	0.14	**0.28**	0.14	**0.24**	0.10
2.6.30-post	0.28	0.14	**0.33**	0.12	0.20	0.12
2.6.31-post	0.18	0.16	0.16	0.15	**0.23**	0.09
2.6.32-post	0.29	0.13	**0.31**	0.19	**0.29**	0.10
2.6.33-post	0.23	0.12	0.22	0.11	**0.30**	0.10
2.6.34-post	0.27	0.10	0.22	0.14	**0.27**	0.12

Table 3 Percentages of files increasing (i.e., “INCR”) or decreasing (i.e., “DECR”) their complexity, clustered in time slots

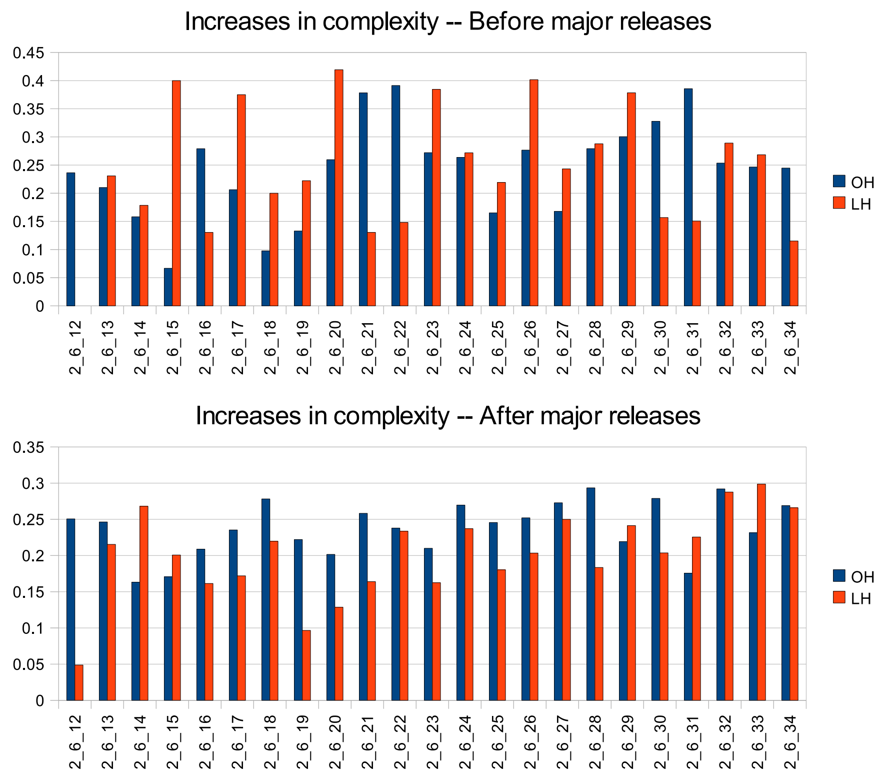

Fig. 9 Portion of files increasing their overall complexity during Office Time (OT) and at Late Night (LN) divided by time slots

Any FLOSS system will need to be studied to extrapolate the appropriate weights to evaluate the above activities. In the study of the Linux kernel, the extrapolated weights are available in Table 4.

Pre-week activity	OH	0.24	0.13	0.63
	AO	0.21	0.11	0.68
	LN	0.24	0.11	0.64
Post-week activity	OH	0.24	0.13	0.64
	AO	0.21	0.14	0.64
	LN	0.20	0.12	0.68

Table 4 Weights to complexity

5 Threats to Validity

This paper has analyzed the Git repository offered by the Linux Kernel community. One of the main reasons for doing so is because this source code management system offers extra information about the date when the actual author [7] and the committer submitted the changes. Like any other empirical study, the validity of ours is subject to several threats. In the following, threats to internal validity (whether confounding factors can influence your findings), external validity (whether results can be generalized), and construct validity (relationship between theory and observation) are illustrated.

1. Internal Validity – the following threats have been detected:
   - In a common working day, there are main differences among developers. Some of them could work in office, but some others could work some time during the mornings, and some more time during the evenings.
   - Our methodology can not be applied in SCM such as CVS or Subversion since they store the time when the commit was submitted, but not when the change in the source code was done by the author (even by the committer).
   - We still need to check how different commands from the Git repository work. In a common way of working, (*pull, merge or push*) there should not appear any problem related to the real authorship and date of a change. However, we still need to study the behavior of commands such as *git cherry-pick* or *git rebase*.
   - In other occasions we could find people traveling around and not changing their timezone in their computers what could add some noise to the data. In other words: some people could work on a different timezone that they really are.
2. External Validity – we have focused our analysis in the Linux Kernel community and also in the Git SCM. Some other FLOSS communities are using other SCM systems which do not store information related to the time when the change was done (in terms of real authorship).
3. Construct Validity – the following threats have been detected:
   - the results of this paper assume that people in different countries work in the same way: of course this should be discounted in several ways, for instance considering that the holiday systems in different European countries and in North America are vastly different, and both are culturally very different from the holiday schemes in other countries in Asia or Africa. This could, in somehow, distort the results, albeit in the case of the Linux Kernel community, they seem to show a common *office* patterns, which facilitate the analysis of the data.
   - Also, We have not taken into account if the changes were made over the source code or were not. A more deeper analysis could show more accurate results with this respect. Since we are measuring activity in the source code, we have studied the SCM system used by the Linux Kernel community, but it

[7] Using the option *–pretty=fuller*

could contain specific files such as text files which are modified, but not being source code.

6 Barriers of Repeatability

Mining software repositories is a task complex in time, but also in tools and datasets used for retrieving information. Some authors [**?**] have addressed the necessity of specify three main questions, which are depicted next for this study (availability of raw data, processed dataset and tools or scripts). The rest of the process have been explained in detail in section 2.2 and 2.3. Thus, this section aims to fill the gaps among the different steps of the method followed to achieve the different developed metrics.

– Raw Data. The raw data used for this paper is the publicly available data sources found for the Linux Kernel community. And more specifically the source code management system that can be found at the Git repository. The dates used for this data are those commits available between the dates *2005-04-16 15:20:36* and *2010-06-29 10:42:52*. This can be easily downloaded by means of the *git clone* command line [8]
– Processes data. All the processed data can be found in a MySQL database and publicly available at `http://alcachofo.libresoft.es/jsme2010-effort-linux/cvsanaly_kernel26_git.mysql.zip`. This dataset has been obtained using the tools and scripts described in next bullets. All the tables were retrieved by the CVSAnalY tool except: *release_commits*, *release_dates*, *compare* and *changes*. With respect to the tables release_dates and release_commits they were both manually introduced to make easier the analysis of the data and they were based on data obtained from the distribution website at `http://www.kernel.org/pub/linux/kernel/v2.6/`. While the other two tables contains information automatically retrieved by the use of some scripts specifically created for this purpose.
– Tools and scripts.
  – CVSAnalY: This tool can be found at `git.libresoft.es` and downloaded using the *git clone* command. The specific version comes from the current version at the master branch found at the date of *2010-08-27*.
  – Scripts: Several scripts have been used to retrieved specific data for each of the questions and charts provided in this paper.

7 Conclusions and Further Work

Recently a well-known and accepted model has been proposed and thoroughly discussed in order to cluster FLOSS developers into the so called “onion model”, where

[8] git clone `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git`

different layers correspond to an increasing productivity and responsibilities. Transversal to such clustering, this paper has approached the issue of characterizing the FLOSS development from the point of view of the time slots of contributions. FLOSS developers are known to be active in various parts of the day and week, unlike a traditional 9am-5pm model of in-house software development. The Git SCM technology provides an advance to such requests, since it allows to properly determine when a developer issued a commit command at her timezone, rather than losing such information by using the SCM server local timezone. Taking into account these results, they could be useful in the field of software estimation costs and effort. SCMs such as Git offer nowadays extra information from the people involved in the FLOSS communities. This helps to calculate the real time when a change was actually submitted. So far, most of the best known SCMs just store the information when a change was submitted to the server, what implies that the real date is missed.

The study on the activity detected in the Linux kernel were compared with what found in the previous analysis of an Agile commercial system, and it became clear that the traditional 9am-5pm development time only accounts for some 55% of the overall activity within Linux: other two time slots were used to characterize the FLOSS development, the period between 5pm and 1am (After Office slot), responsible for some 31% of activity and the period between 1am and 9am (Late Night slot), responsible for some 14% of overall activity. An effort estimation model would therefore to take into account such distribution of activity to properly model a FLOSS development, by firstly estimating the weights of the various time slots.

The study of productivity within the Linux kernel showed that a positive bias is imposed when a major release is due: the analysis of added, deleted and modified lines shows regularities when considering only the weeks before and after a major release. An increased productivity is always detected in all the measured attributes after a major release, which calls for an updated model of effort and productivity estimation both before and after a major release.

Finally the study of code quality has shown that time slots should also be considered as differently contributing to the overall complexity within a project: it was indeed found that the Late Night and After Office slots should be carefully monitored since they more often introduce additional complexity both in the weeks before and in the weeks after a major release. An effort estimation model was developed to take into account such time slots and the presence of a major release, that can be generalised to any FLOSS, round-the-clock project.

With respect to further work, this work could be useful in the field of cost and effort estimation in FLOSS projects. A better characterization of the commit patterns, such as studying each of the developers by their blocks of activity could improve estimation models, as well as dividing the effort in the various parts of the day. For instance, if a committer is usually working during the *office time* and she usually submits a change every two hours, we could suppose that she has been working for the whole day around eight hours. Some other patterns could show activity during the weekends. For example, some developers could submit some changes just during specific days. We suspect that this kind of patterns is totally different from the aforementioned one. In fact, in this case, we should measure the real effort in other terms and only taking into account that day.

References

1. R. W. Wolverton, “The cost of developing large-scale software,” *IEEE Trans. Comput.*, vol. 23, pp. 615–636, June 1974. [Online]. Available: http://portal.acm.org/citation.cfm?id=1310173.1310916
2. B. W. Boehm, *Software Engineering Economics*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.
3. K. Molkken and M. Jrgensen, “A review of surveys on software effort estimation,” in *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ser. ISESE ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 223–. [Online]. Available: http://portal.acm.org/citation.cfm?id=942801.943636
4. J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona, “Effort estimation by characterizing developer activity,” in *Proceedings of the 2006 international workshop on Economics driven software engineering research*, ser. EDSER ’06. New York, NY, USA: ACM, 2006, pp. 3–6. [Online]. Available: http://doi.acm.org/10.1145/1139113.1139116
5. A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and mozilla,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, 2002.
6. K. Crowston, B. Scozzi, and S. Buonocore, “An explorative study of open source software development structure,” in *Proceedings of the ECIS*, Naples, Italy, 2003.
7. M. Aberdour, “Achieving quality in open source software,” *IEEE software*, pp. 58–64, 2007.
8. F. P. Brooks, Jr., “The mythical man-month,” in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, p. 193.
9. P. J. Adams, A. Capiluppi, and C. Boldyreff, “Coordination and productivity issues in free software: The role of brooks’ law,” in *ICSM*. IEEE, 2009, pp. 319–328.
10. A. Capiluppi, J. Fernandez-Ramil, J. Higman, H. C. Sharp, and N. Smith, “An empirical study of the evolution of an agile-developed software system,” in *ICSE ’07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 511–518.
11. C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu, “The promises and perils of mining git,” in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*. Citeseer, 2009, pp. 1–10.
12. V. R. Basili, G. Caldiera, and D. H. Rombach, “The goal question metric approach,” in *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994, pp. 528–532, see also http://sdqweb.ipd.uka.de/wiki/GQM.
13. G. Robles, J. M. Gonzalez-Barahona, D. Izquierdo-Cortazar, and I. Herraiz, “Tools for the study of the usual data sources found in libre software projects,” *International Journal on Open Source Software and Processes*, vol. 1, no. 1, 2008.
14. T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, pp. 308–320, December 1976.
15. T. J. McCabe and C. W. Butler, “Design complexity measurement and testing,” *Communications of the ACM*, pp. 1415–1425, December 1989.
16. G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, “Analyzing cloning evolution in the linux kernel,” *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.
17. M. Godfrey and Q. Tu, “Evolution in open source software: A case study,” in *Proceedings of the International Conference on Software Maintenance*. Citeseer, 2000, pp. 131–142.
18. ——, “Growth, evolution, and structural change in open source software,” in *Proceedings of the 4th International Workshop on Principles of Software Evolution*. ACM, 2001, p. 106.
19. C. Izurieta and J. Bieman, “The evolution of freebsd and linux,” in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, p. 211.