Perpetual Development: A Model of the Linux Kernel Life Cycle

Dror G. Feitelson

School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
Tel: +972 2 658 4115; email: feit@cs.huji.ac.il

Abstract

Software evolution is widely recognized as an important and common phenomenon, whereby the system follows an ever-extending development trajectory with intermittent releases. Nevertheless there have been only few lifecycle models that attempt to portray such evolution. We use the evolution of the Linux kernel as the basis for the formulation of such a model, integrating the progress in time with growth of the codebase, and differentiating between development of new functionality and maintenance of production versions. A unique element of the model is the sequence of activities involved in releasing new production versions, and how this has changed with the growth of Linux. In particular, the release follow-up phase before the forking of a new development version, which was prominent in early releases of production versions, has been eliminated in favor of a concurrent merge window in the release of 2.6.x versions. We also show that a piecewise linear model with increasing slopes provides the best description of the growth of Linux. The perpetual development model is used as a framework in which commonly recognized benefits of incremental and evolutionary development may be demonstrated, and to comment on issues such as architecture, conservation of familiarity, and failed projects. We suggest that this model and variants thereof may apply to many other projects in addition to Linux.

Key words: Software evolution, Software release, Maintenance, Linux kernel

"Linux is evolution, not intelligent design."
Linus Torvalds

1. Introduction

Classical software lifecycle models typically partition the software's life into two periods: development and maintenance. The models focus on the development, further partitioning it into phases and articulating its iterative nature. Maintenance is typically not discussed in much detail, despite the wide agreement that it is vital and consumes considerable resources. It is simply stated that the same principles and techniques used in development continue to apply, and that actually all of development is done so as to facilitate maintainability.

The implication of such models is that the full system size is expected to be achieved at the end of development. Only then is the system delivered to the customer and installed in the field. This applies even for very large and complex systems. The customer is expected to have participated in the requirements engineering and in the testing, but is not expected to have actually used the system for real. Once the system is installed, maintenance involves both correction of faults (corrective maintenance) and updates and improvements (adaptive and perfective maintenance) — but is not expected to involve any major additional development. As the term “maintenance” implies, the goal is essentially to extend the usability of the system, not to enlarge it.

Comparing this approach with those underlying other fields, we find a strong match with the construction of large physical systems. For example, an ocean liner, a shopping mall, a new airport, or a cathedral are designed and built with inputs from the customer who initiated them, but they cannot be used before completed. Once completed, they are used and maintained for an extended period. The construction and maintenance are clearly distinct periods with different characteristics.

But there are other large systems that are built in a completely different manner, both in the physical world and in cyberspace. These are not created from scratch in one fell swoop, but rather evolve from humble origins over a long period of time. In particular, evolution is often seen in new technologies, even when we like to think of the changes they spawn as revolutionary. Consider automobiles as an example. The first step in producing automobiles was to replace a horse with a motor, while maintaining the overall structure. Gear boxes and improved suspension came later. So did batteries for starting the motor and accessories such as a radio receiver. Thus while each individual car was developed and then maintained, the overall dynamics were of evolution from one year to the next. Each step improved the usability of the new cars, and enabled a larger following to emerge. This happened in parallel with the development of an infrastructure for gasoline distribution and an improved and incrementally growing road system — neither gas stations, roads, nor cars would be justified without the others.

A similar process may underlie the evolution of dynamic social systems — a university, a multinational corporation, the stock exchange, or a bazaar, for example. Such systems are the product of a long process of give and take that refined their details, so their evolution is guided by continuous interactions between the system and its environment. This interaction is extremely important, as it is the impetus for further developments and guides the way towards developments that will actually be useful in practice. Thus the system is never “designed” in the normal meaning of the word, and its requirements are not and can not be specified in advance. Nevertheless, this is the easier and safer path to follow, as having to design everything in advance can be a lot of work and may lead to great risks if something is not done right.

Large software systems also often evolve in this manner, possibly in parallel to the evolution of the social or business systems that use them. For example, the software systems underlying the operations of Amazon.com, Ebay, and Facebook are much more complex and feature rich now then they were when these companies were founded. But this was not designed in advance, and the specific features that would emerge could not be anticipated. Instead, the software evolved and grew together with the company, an epitome of Lehman’s “E-type” systems [38]. A similar process can also happen with software that has a general user community rather than being used by a specific company. In this paper we focus on modeling the evolution of one such system, namely the Linux kernel.

Evolutionary development has been recognized as an important paradigm at least since the seminal works of Lehman [39] and Gilb [21]. In fact, is can be argued that today most of the software industry follows evolutionary principles at least to some degree. This is especially marked in software product lines, where successive software products explicitly follow and extend each other. It is also the underlying basis of the agile software movement. However, evolution can in principle take many forms. For example, Lehman's first law of software evolution was that evolving systems change continuously. This implies that while new features are added, old ones should be removed at a similar rate. His sixth law, formulated nearly a decade later, was that evolving systems continuously grow. This implies that change is actually the result of many more additions than deletions. In retrospect this appears to be the more common approach, and indeed this is what we observe in Linux [51, 70].

In software engineering terms the process of evolutionary development can be viewed as extending the notions of iteration and incrementation, with actual production use by real users as an integral part of the loop. The novelty of our model is that each cycle is not viewed independently, but rather the continuation from one cycle to the next is emphasized. One of the byproducts of this point of view is the deprecation of the terms "delivery" and "maintenance". Instead, the software continues to evolve and grow in a process of *perpetual development*, with regular releases of new production versions replacing the singular delivery of the finished product. This is indeed what we find for the Linux kernel, and also in many other (mainly open source) projects. Another byproduct is exposing the relationship between parallel versions of the project that exist at the same time. In particular, maintenance is applied to production versions that exist in parallel to the main development version.

Our main goal in this paper is to articulate the perpetual development model, using the Linux kernel as a motivating example. Using the Linux data, we demonstrate the central role of the growth of the system over time, the continuous nature of development and release activities, and how they differ from maintenance activities. By providing a detailed analysis of Linux development, we attempt to uncover development patterns that are potentially widely used, but have not been articulated and studied in the literature. We then briefly speculate on the implications of these development patterns.

The choice of Linux is not accidental. Linux is perhaps the best-known and most successful open-source project in the world, to the point of becoming the poster-child of the whole open source movement. The source code of its full development history is freely available from www.kernel.org and numerous mirrors worldwide — at present totaling 1322 versions (Table 1 and Table 8). Importantly, Linux's development defies common management theory [60]. At the same time, it does not fit into any common software lifecycle model. This motivates us to suggest the perpetual development model. This model is descriptive rather than prescriptive. Its goal is to create a framework for discussing what is actually being done in the field and why, rather than attempting to dictate ideals. In particular, we note that Linux development is a dynamic process in constant flux, and adapts to overcome problems that occur as the system grows. Thus different variants of the model apply to different parts of Linux's evolution.

The rest of this paper is organized as follows. In the next section we review related work on

designation	*number*	*versions*
production	159	1.0, 1.2, 2.0, 2.2, 2.4
development	397	1.1, 1.3, 2.1, 2.3, 2.5
pre-release	56	pre2.0, 2.2.0-pre, 2.3.99-pre, 2.4.0-test, 2.6.0-test
2.6 production	463	2.6.*
2.6 release candidates	247	2.6.*-rc
total	1322	

Table 1: *Linux versions and their designations. Only full versions are counted, ignoring patches that were used especially in the early years.*

modeling the development of long-lived evolving projects. The perpetual development model is then presented in Section 3. Section 4 elaborates on releases, which are of central importance in the model. This is followed by a detailed discussion of Linux's growth in Section 5. Finally, we present some possible implications of the model in Section 6 and conclude in Section 7.

2. Lifecycle Models for Long-Lived Software Projects

Classical software lifecycle models typically cover the development from a concept to a delivered software product. A recurring feature in early models such as the waterfall model and the V model was the quest for stability. First, one needs to get all the requirements right. This is then used to formalize comprehensive specifications. Given the specifications, we can create a design that satisfies them, and so on. But this scenario is often simply irrelevant in real life, because the clients can't articulate all their requirements in advance, so specifications are never complete, leading to designs that will have to change when more requirements are discovered [49].

The alternative is to consider development as a learning process, where requirements and specifications grow together as more experience is gained and risks are understood. This is incorporated into models by employing an iterative and incremental process [36]. In principle, this can continue beyond the initial delivery of a working system to its users. Indeed, Boehm mentions the applicability of the spiral model — complete with risk assessment at each stage — to proposed enhancements that may be applied to systems already in operational use [5]. Likewise, Kruchten suggests the application of additional "generations" of the unified process to handle system evolution [35]. But their main focus remains the basic initial development.

As a result of focusing on the initial development, up to product delivery, common lifecycle models do not apply to the full life-span of long-lived software products. In particular, they do not describe the relationship between successive releases of the product. This has prompted the development of specialized lifecycle models to fill this gap.

One such model is the evolution-tree model of Tomer and Schach [67]. The levels of this tree represent stages of development: requirements, analysis, design, and implementation. The evolution of a software product is then described as successive branches of the tree, where each new version backtracks and then branches out from some level in the previous branch. Thus if only the implementation is modified relative to the previous version, the new branch will start at the third level, but if there are new requirements the new branch will start directly from the root. At the same time, development artifacts such as design documents may be reused across corresponding levels in successive branches.

Another model is the staged model of Rajlich and Bennett [59]. This model creates a framework for describing the evolution of a software product, with each version passing through the stages of evolution, servicing, phaseout, and finally closedown. Notably, the "versioned" variant of the model allows for multiple serviced versions branching off from the main sequence of evolved versions. A system that cannot continue to evolve, but is still serviced, turns into a legacy system. This model provides two important features that are lacking in the Tomer and Schach model. First, it places a much stronger emphasis on the continuity of the whole process, as new versions always start from a previous one without backtracking. Second, it allows for the parallel existence of multiple versions, and in particular, for new development that is done in parallel with the maintenance of an existing production version. Thus it decouples evolution from maintenance, and avoids the rat-hole of arguments whether these terms are synonymous [22]. A related model is the split and re-integrate model of Nakakoji et al. [53]. This may be viewed as a refinement of the Rajlich and Bennett model, where developments done in one branch are propagated to another branch that is developed in parallel. Capiluppi et al. have observed that in open-source projects a legacy system may be picked up and revived by a new set of developers, thus returning it to the evolution phase [9].

Our model is also a refinement of this approach, which adds another dimension: how the product grows during its evolution. In addition, we emphasize the implications of continued development and parallel branches, as opposed to the managerial aspects as emphasized by Rajlich and Bennett. Thus we do not deal with issues like staffing, choice of language, change of technology, etc. The model can also be viewed in relation to Raymond's works on open-source development (also inspired by Linux) [60], where we try to formulate a lifecycle model to his technical observations and description of social processes.

Indeed, Linux evolution has been studied by several authors. Several authors have studied Linux growth, especially in connection with Lehman's Laws of software evolution [24, 61, 28, 31, 30]. This is also related to the potential increase in complexity, and the ensuing consequences regarding maintainability [64, 63, 65, 30]. Others consider mechanisms of evolution, and especially the use of cloning [23, 51, 2, 54, 46]. Perhaps the closest to our work is the paper by Godfrey and Tu [24]. In particular, our rendition of Linux growth and parallel versions in Fig. 2 extends graphs drawn by them, as does the discussion of individual subsystems in Section 5.3. But all these works typically do not formulate an explicit lifecycle model.

3. The Perpetual Development Model

Conventionally, lifecycle models describe the sequence of actions that need to be taken and the transitions between them. The perpetual development model is more about continuous activities, and in particular, activities that happen in parallel. However, it also includes singular decision points where activities change or branch off from each other.

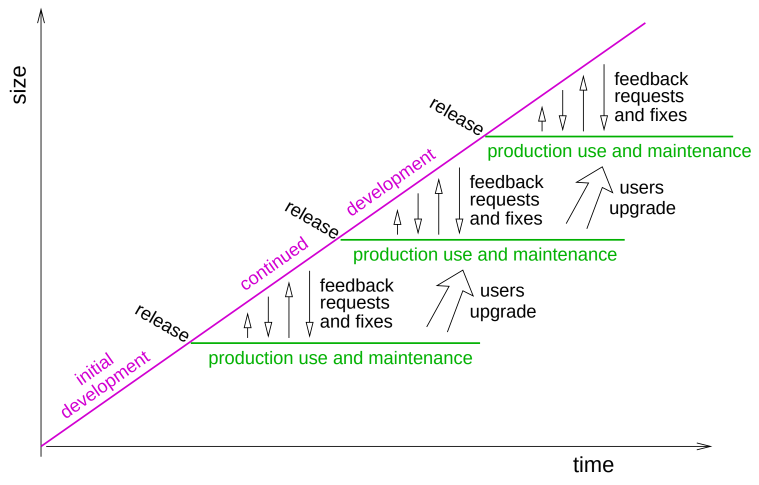
size
time
initial development
release
continued development
release
release
production use and maintenance
production use and maintenance
production use and maintenance
feedback requests and fixes
feedback requests and fixes
feedback requests and fixes
users upgrade
users upgrade

Figure 1: *Basic version of the perpetual development model.*

3.1. Modeling Growth and Parallel Activities

Practically all lifecycle models use 2D pictures for illustration. The waterfall is typically drawn as a diagonal sequence of blocks, possibly with some other elements, where the direction from top-left to bottom-right correlates with some notion of progress and time. The spiral model, as its name suggests, is drawn as a spiral, where the angle reflects the sequence of activities within each cycle, and the radius reflects the cycle number. In the models of Tomer and Schach and Rajlich and Bennett both dimensions correlate with time, one depicting progress within a version and the other progress from one version to the next. However, neither dimension explicitly reflects time with a linear scale.

The perpetual development model explicitly uses the two dimensions to portray time (on the X axis) and size (on the Y axis). “Size” can be interpreted literally, or alternatively it can be interpreted as the feature set provided by the software. As quantifying features may be harder to define, we limit the discussion to physical size.

Given these axes, the model consists of a continually growing *development backbone*, from which stable *production versions* branch out at intermittent *release points* (Fig. 1). Thus several versions of the product may exist simultaneously. In this framework, the following activities occur:

1. Users of production versions provide feedback and new requirements to developers, and bug alerts to maintainers.
2. Maintainers maintain the current production versions. This may also involve interactions with developers working on the next version.
3. Developers, at the same time, continue to develop the system. In doing so they use the input from the users and maintainers of the current versions.
4. The development activity is punctuated by releases of new versions. Developers therefore alternate between two types of activity: implementing new features and performing a release. We expand on this issue in Section 4.

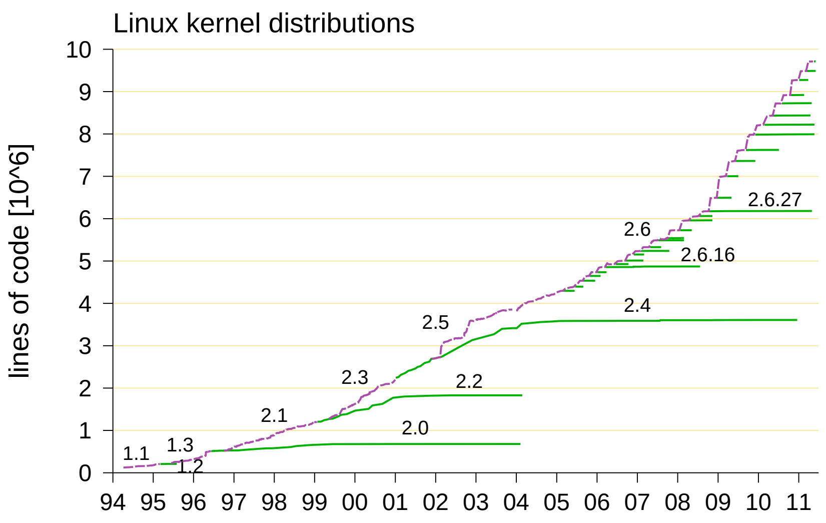

Figure 2: *Perpetual development of Linux from 1994 to 2011. Up to 2003, versions with odd major number (1.3, 2.1, etc.) were development versions, and those with even numbers (1.2, 2.0) were stable production versions. In version 2.6 only production versions are released, while new development work is reflected only in release candidates.*

5. When new production versions are released, users may upgrade to the new version. This may be a slow process, and not all users necessarily upgrade.

It is important to be precise regarding the meaning of the term “release”. In the model we use this term exclusively to mean the release of a new production version of the system — an event sometimes referred to as a “major” release. We distinguish this from the “minor” releases that occur when a new instance of an existing version is made available. In production versions such new instances typically reflect some maintenance work that has been done, e.g. the correction of a bug or the application of a security patch; in development versions, they may reflect additional progress in the development. To avoid ambiguity, we shall refer to such minor releases as “updates”. And when we refer to a Linux kernel version, such as 2.4, we are actually referring to a series of such minor versions, of the form 2.4.x.

The model and the list of parallel activities are based on observation of the development of the Linux kernel from its initial release in 1994 to mid 2011, as depicted in Fig. 2 [30]. For size we use lines of code; this and other possible size metrics are discussed below in Section 5.1. The growth of the backbone need not necessarily be strictly linear as in Fig. 1, and indeed the Linux data exhibits some fluctuations and a generally increasing growth rate [24]. This is discussed in Section 5.2. The production offshoots remain relatively stable as may be expected (the initial growth of 2.2 and 2.4 is discussed below).

An important feature of the model is to demote the status of product delivery (as was also done in the long-term lifecycle models cited above in Section 2). Delivery is no longer a focal point of the process, dividing the product lifetime into development and maintenance. Rather, it becomes one of many similar points along the continuous process of development, where new production versions are released. When such new releases are made, users start to upgrade to the new production version at their own pace, but some may choose not to upgrade at all. Production versions therefore generally need to be maintained well beyond the release of the next version. For example, Linux kernel version 2.4, which was first released in January 2001, was still being maintained in late 2010. More recently, versions 2.6.16, 2.6.27, and 2.6.32 were each maintained for several years. In fact such behavior is not unique to Linux. A notable example is the release of Windows Vista, where many users elected not to upgrade but rather to keep using Windows XP.

Another important feature of the model is to delimit the scope of maintenance activities. Maintenance implies an effort to preserve and sustain the usability of the system, possibly with peripheral modifications, but it does not involve continuing to build the core of the system. This distinction between maintenance and continued development, which is central to our model, is missing from many discussions of maintenance, which implicitly assume there is only one version of the product (e.g. [32]). It also relieves the forced inclusion of continued development in "perfective maintenance" [45], which is necessary if one insists on using the "maintenance" label for everything that happens beyond the first delivery. Thus, while maintenance does in fact apply to versions of the product that are in production use, it is not the main activity but rather done in parallel to continued development of the backbone. And from a terminology aspect, maintenance is not a synonym for evolution.

An interesting question that arises from the distinction between continued development and maintenance is who performs these activities. In the context of open-source projects, maintainers and developers may be the same people, as developers may have ownership or at least a feeling of responsibility for their code, and will therefore maintain it in parallel to performing other development activities. Thus one may expect maintenance activities to be strongly correlated with a subset of development activities that happen at roughly the same time. For example, bugs found during continued development may point to similar bugs in parallel production versions, and vice versa. Moreover, users may also be the same people, as the community of kernel hackers largely develops and maintains Linux for their own use. One may also extend this to the "real" end users, which are mainly Linux distribution companies like Red Hat, Debian, and Novell/SUSE, who nevertheless contribute significantly back to the community[1].

The graphical representation of perpetual development also provides a framework for demonstrating the known benefits of iterative and incremental development with evolutionary delivery, such as [21]

- The lead time to the first working software is short. Thereafter, a working version always exists for the benefit of users. Thus there is little danger of the project coming to nothing.
- Real users doing real work are effectively brought into the development loop. Their work acts as a test of system functionality, and helps in uncovering problems [60].
- Having actual users work with the system also uncovers new requirements that were not

[1]Canonical, the company behind Ubuntu, seems to focus more on desktop software than on the operating system kernel.

anticipated in advance, and allows for prioritization of different requirements that have not been implemented yet. The user requirements and the system that solves them co-evolve together [38, 48].

- The relatively short time between releases implies focus and limited scope. This is the evolutionary alternative to progress in the "cone of uncertainty" [4]: rather than starting with wide uncertainty and working to reduce it, maintain a much smaller scope (and hence smaller uncertainty) throughout.
- The relatively short time between releases also implies that prioritization and bounding becomes crucial: at each step, one needs to decide what to do and what *not* to do, based on user input or on the effect on the bottom line [16].

In addition, it highlights issues that are often not noticed, such as

- The danger of releasing an unstable version is reduced, because users upgrade to the new version only gradually, and they have the previous version as a fallback.
- The continued growth of the system implies that new code is generated all the time. This calls into question the notion that eventually incremental change and refactoring will become the dominant activities rather than "clean" development from scratch [58].
- The long time scale and recurrent nature of the project implies that new technology may be incorporated in the development process as it becomes available, as opposed to freezing everything at the outset.

The novelty of the perpetual development model is mainly in articulating the continuous and parallel aspects of software development that are often left implicit or beyond the scope of the lifecycle model. The difference from common models based on iteration and incrementation is that the process is not expected to end with a "final" release [4] — it just goes on and on as long as it is useful. Of course it will most probably end eventually, but the mindset is one of perpetuation. We allege that this mindset is not unique to Linux, and is often present in open-source projects [60] and in agile development [3].

3.2. Relationship with Agile Development

The perpetual development model may seem to be redundant due to its similarity with evolutionary development in general [21], or agile development in particular [3]. However, these terms actually denote different things.

As described in the introduction, we first make the distinction between projects which undergo evolutionary development and projects which are developed and then maintained (Fig. 3). This distinction relates to the nature of the project. Evolutionary development is suitable in situations where the rate of change is high, and the project requirements cannot be defined in advance. A develop-maintain style is suitable for those situations where one must define the full project in advance. In such cases, the learning process inherent in the evolutionary approach must be replaced by careful planning and modeling. Indeed, this was the impetus for the development of tools like statecharts [26]. These were developed as part of an effort to nail down the requirements for the

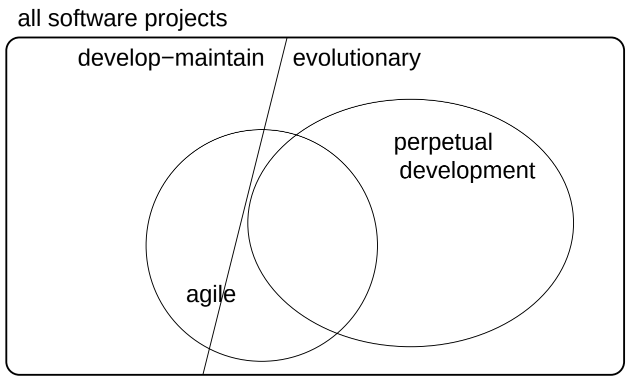

Figure 3: *Relationship between evolutionary software development, perpetual development, and agile development.*

software controlling a new fighter plane, a good example of a situation where essentially all the software is needed for an operational system, the system does not change much after initial delivery (except for new versions for new models of the plane, similar to a product line), and it is practically impossible to carve out a small initial chunk that would be useful in itself.

Perpetual development is one approach to accomplish evolution, namely evolution by continuing to develop the product. This implies continued growth, as reflected by Lehman's 6th law. Note, however, that growth is not necessarily implied by the term "evolution" by itself. In the animal kingdom, for example, there are some cases where evolution involved growth in size (think dinosaurs, giraffes, and whales), but many where it did not (e.g. bacteria and other single-cell organisms). For software it is also possible to envision evolution by changing and adapting the codebase, rather than by adding to it. However, it seems that in practice perpetual development is a common approach to software evolution, and perhaps the dominant one.

The dictionary definition of "agile" is something that is light and quick. In software development, this term has come to mean an iterative process with little if any long-term planning[2]. Still, many agile projects are terminal projects, where the agile approach is followed for the duration, but it is expected to end either when all the desired features are done or when the allotted time (and possibly budget) is exhausted. At the same time, there is an emphasis (made explicit as one of the core practices of Extreme Programming, for example) on maintaining a "sustainable pace" of development that can continue indefinitely. Perpetual development focuses on this aspect, and highlights the continuous nature of the work as an enabler of all the other features. This is in contrast to most interpretations of agile development, which emphasize the contrast with "death march" projects and overtime, and the importance of technical elements such as the lack of formal detailed planning and the use of pair programming.

Furthermore, a major difference is that perpetual development is a lifecycle model, whereas agile development is a methodology. Thus instantiations of agile development, such as Extreme Programming or SCRUM, each promote specific procedures and practices. Examples include

[2]See The Agile Manifesto, available at http://www.agilemanifesto.org/, for guiding principles.

maintaining a backlog of features, conducting daily standup meetings, and adhering to a strict release schedule. Our case study of Linux is a good example of this distinction: Linux is definitely an instance of perpetual development, but it does not subscribe to any specific agile methodology. Thus, while it currently adheres to a regular release schedule with a cycle of 2–3 months, this was not always the case, and it is not part of the definition of the model.

3.3. Variants in the Linux Case Study

The Linux data shown in Fig. 2 is similar to the model introduced in Fig. 1, but not identical to it. Indeed, the model attempts to gloss over differences, and present the most salient concepts in a clean manner. Reality is often more complex.

It is easy to identify three phases in the Linux kernel's development (as reflected in Fig. 2), which can be viewed as variations on the more abstract and general model. The first is the version 1 kernels, from 1994 to 1996. During this period most of the activity was in development, and only few maintenance updates were made to production versions. The second phase, from 1996 to 2004, is characterized by the release of three long-lived production versions (2.0, 2.2, and 2.4) that were maintained in parallel. Significant developments were performed between these versions. This led to big differences between them and long intervals between their release dates which were extended even further by the protracted release process itself. Such long intervals contradict the desired rapid release cycle of open source software [60]. Indeed, in the third phase (kernel version 2.6 since 2004) production kernels are released regularly every 2–3 months. To reduce the maintenance effort most of these are not maintained much beyond the release of the next production version.

An especially troubling aspect of the Linux data is the significant growth observed in the initial periods of the 2.2 and 2.4 production versions. This contradicts the assumed role of these versions, where production versions are only maintained and not expanded with new developments. The explanation is that new developments were indeed originally injected into the development versions (specifically 2.3 and 2.5). However, due to the long delay expected until the release of the next production version, many new developments were then propagated into the existing production versions [24]. The switch to the more rapid release cycle of 2.6 is perhaps a reaction to this state of affairs, and an attempt to institutionalize a more orderly process for the quick dissemination of new developments.

In summary, the Linux data exposes two main variants of the perpetual development model. In one the production releases are far between, and contemporaneous development and production versions are coupled together. In the other production releases are frequent, and production versions are largely decoupled from development. This distinction is related to the distinction between releases based on features and releases based on schedule. In the next section, we show that the mechanics of the releases themselves are also different.

4. Releases and Decision Points

As noted above, the focal points of the perpetual development model are the major release points where new production versions are released. These define the structure of the system's

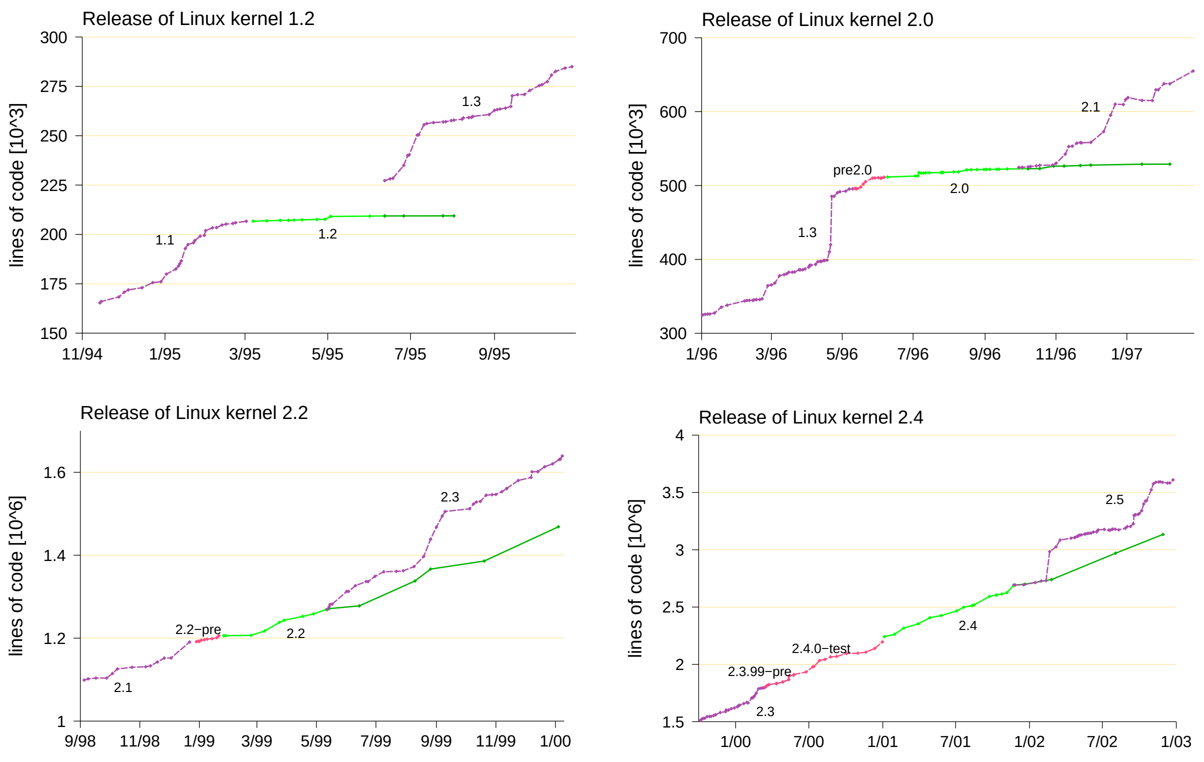

Figure 4: *Releases of Linux kernel versions 1.2, 2.0, 2.2, and 2.4.*

development and the relationship between its branches. It is therefore of interest to investigate the process of performing such a release in some detail. In Linux, this refers to the releases of the major production versions (1.2, 2.0, 2.2, and 2.4), and to the "third digit" releases of the 2.6 versions starting with 2.6.11. As we show, these two sets of releases are rather different from each other.

Note that we do not refer to updates (or "minor releases") of existing versions — the third-digit releases before version 2.6.11, and the fourth-digit releases of 2.6.11 and later. In Linux, such updates are made when "enough" content accumulates or when there is a new security patch that needs to be disseminated quickly. This is a subjective decision made by whoever is responsible for the version in question.

4.1. Idealized Release Model

Based on the Linux case study we can take a more detailed look at the activities surrounding releases of new production versions. We initially focus on the four major production versions: 1.2, 2.0, 2.2, and 2.4. A zoom into the update activity surrounding each of these releases is shown in Fig. 4. This indicates that a release is not a point, but a whole sequence of activities in itself.

While not identical, these four releases exhibit the same general structure. Using the release of kernel version 2.0 as an example, the last update of version 1.3, which was 1.3.100, took place on 10 May 1996. On 12 May 1996 version pre2.0.1 was released, indicating the beginning of work

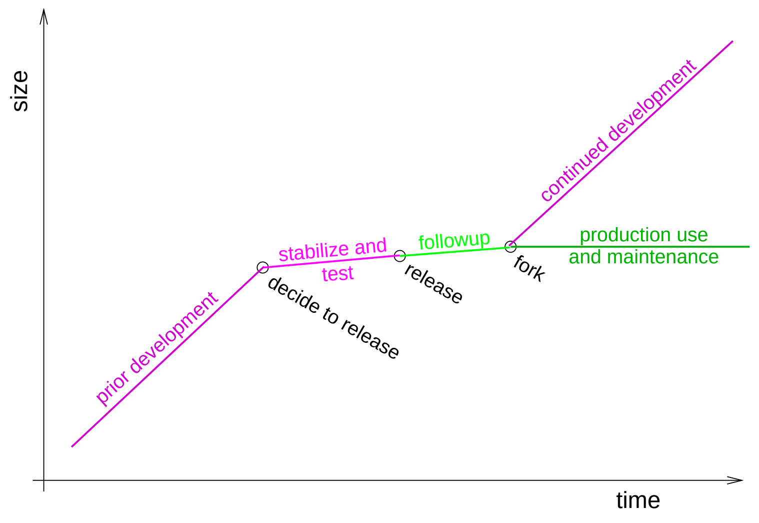

Figure 5: *Idealized model of the anatomy of a release.*

towards the release of the 2.0 version. There were a total of 14 updates in the pre2.0 series, ending with pre2.0.14 on 6 June 1996. Three days later, on 9 June 1996, version 2.0 was released. The fork of a new development version took place on 30 September 1996, with the release of kernel version 2.1.0. By that time, 21 updates of version 2.0 had taken place.

The differences between the four releases are also instructive, and seem to indicate that the release process was refined with time. Thus release 1.2 did not have a separate pre-release series of updates leading up to the release itself. At the other extreme, release 2.4 had two separate series: one of preparing the new version, and another of testing (and presumably correcting) it.

Based on the above, we can generalize an idealized release process that includes the following set of decision points and activities between them (Fig. 5):

1. Decide to release. The decision at this point in time is to stop developing new functionality, and prepare to release what has accumulated so far. Except for version 1.2, this is reflected by a new pre-release series of kernel updates. The following activity is one of stabilizing and improving the code, based on internal testing by the developers. Additional development is done mainly to fill holes in existing functionality, not to add new functionality, so the rate of growth is expected to be reduced.
2. Perform the release. This is the actual point of the release itself, where the new production kernel is released and the series of its updates is started. The initial updates reflect a period of support and followup, in which developers continue to stabilize and improve the code, but now this is based on feedback and bug-reports from actual users. Again, this may include some additional development.
3. Fork a new development version. This decision signifies that the released version is considered stable and usable. It is reflected in the data by the forking of a new development version, which will be pursued in parallel to the maintenance of the production version that has just been released.

series	*updates*		*duration*		*result*
pre2.0	14		28		2.0
2.2.0-pre	9		28		2.2.0
2.3.99-pre	9	22	71	296	test1
2.4.0-test	13		225		2.4.0

Table 2: *Stabilization work leading up to the release of production versions in Linux. Duration is in days.*

version	*forked from*	*delay*	*relationship*
1.3.0	1.2.10	97	modified
2.1.0	2.0.21	113	modified
2.3.0	2.2.8	106	identical
2.5.0	2.4.15	322	identical

Table 3: *Forking of development versions from production versions in Linux. Delay denotes the time from the release of the production version in days.*

In addition, there is a final decision point related to production versions:

4. Discontinue maintenance. The decision to stop supporting a production version is usually taken only after the subsequent production version is well established. It does not mean that this version will immediately cease to be used — only that it will not be further maintained, so no additional updates will be made. In Linux the decision to cease maintenance is not necessarily the final word, as distributors (such as Red Hat or Debian) may continue to maintain a version that is important to them even when it is no longer "officially" maintained. For example, Red Hat guarantees support for its Enterprise Linux versions for 10 years from their release date.

In a business setting, there may be additional phases due to financial or legal considerations [59]. This seems to be largely irrelevant for open-source systems such as Linux.

This model of a release is different from the common software release life cycle [71]. In the common model, the pre-alpha phase denotes development, and the alpha phase is testing. This distinction is irrelevant for Linux, as testing, to the degree that it is done, is a continuous activity (similar to the testing workflow in the Unified Process). The "decide to release" point above is essentially equivalent to the decision to perform a beta release in the more conventional software release life cycle. Indeed, Linux also uses the terminology of "release candidate" for such versions. The main differences come with the release itself. The release decision point is semantically equivalent to the common "release to manufacturing" (RTM), meaning that the software is passed from the development unit to the manufacturing unit and reproduced; this subsequently leads to "general availability". In Linux and other open-source software distributed on the Internet there is no manufacturing step, and released software is immediately available. But more importantly, support by the development team continues after release. This followup phase, which continues until the decision to fork a new development version, is missing from the common model.

Data regarding all four major releases is provided in Tables 2 and 3. All show the basic structure

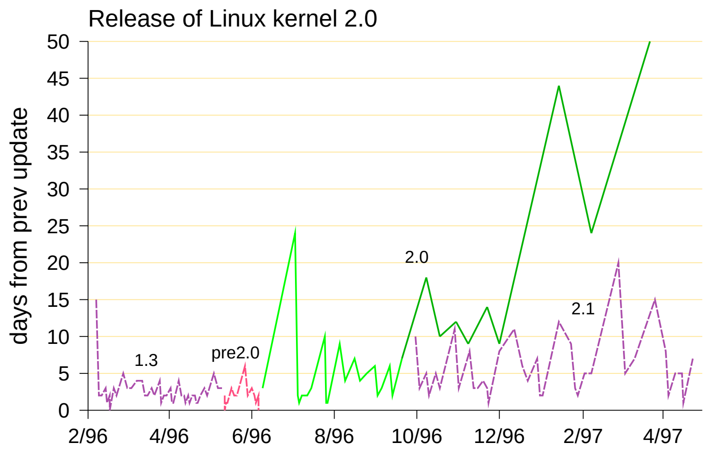

Figure 6: *Update rate around the release of Linux kernel version 2.0.*

described above, albeit with some occasionally major variations. Table 2 shows that the stabilization phase is typically relatively short. Preparing for versions 2.0 and 2.2 took less than a month. In the case of the version 1.2 release there was even no explicit switch to work on stabilization. On the other hand, the work towards version 2.4 was much longer, and took nearly 10 months. This was divided into a bit more than 2 months of preparation, followed by more than 7 months of testing.

Table 3 shows that in all four releases the followup phase was rather long: more than 3 months for versions 1.2, 2.0, and 2.2, and nearly 11 months for 2.4. The first new development version was either essentially identical to the previous production update, or else it already reflected some new developments. As shown in Fig. 4, in both 2.2 and 2.4 there is some growth in the stabilization and followup phases, and continued growth of the production version even after forking the new development version. This indicates that in these versions the distinction between production and development may not be as crisp.

Beyond the structure of these releases, one should notice their protracted nature. The process of preparing a release, testing it, and performing the required followup always took more than 4 months; for 2.4 it took no less than 20 months. Similar problems have been observed in other systems as well [52]. In Linux the solution was to switch to a tightly regulated schedule-based release process, as described below.

Returning to the structure of the release, it is also instructive to observe the rate of work in the different phases, as reflected in their update rate. This is shown in Fig. 6 for the 2.0 release (the other three releases exhibit qualitatively similar behavior). The last months of version 1.3, and the month of version pre2.0, were characterized by a typical rate of a new update every 2–3 days. The first four months of 2.0, and the continued development of 2.1, exhibit a more moderate average rate of releasing a new update approximately every 5 days, with isolated instances of 20 days or more. But once 2.1 was forked, the intervals between updates of 2.0 go up first to around 10–13 days, and then to durations that are better measured in months rather than days. Thus we can see

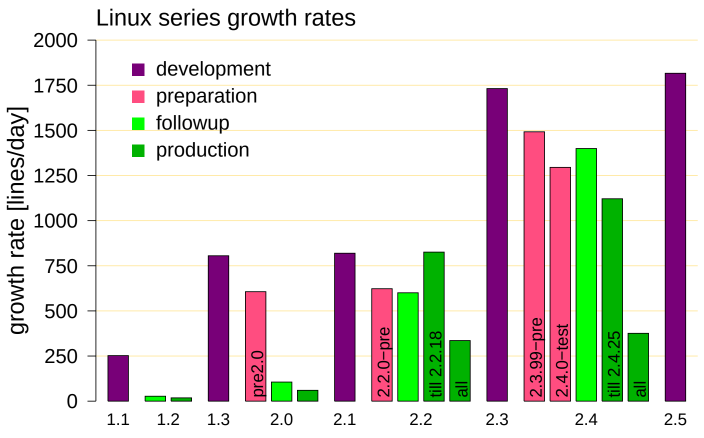

Figure 7: *Growth rates of the different series and their main segments.*

that the initial followup part of a new production release is indeed more similar to development work than to the subsequent maintenance work.

Related to the above is the growth rates exhibited in the different segments. The model in Fig. 5 implies a reduced growth rate in the preparation and followup phases, relative to the development versions. This is indeed the case as shown in Fig. 7 (where growth rate is expressed in lines per day, averaged over the full duration of each phase). For each major release, we see some reduction from the preceding development growth rate, and an increase in the following development growth rate. Thus the sustained work rate (as reflected in updates) is indeed at least partially diverted to activity other than development of new functionality. Even the initial (growing) phases in 2.2 and 2.4 adhere to this pattern. However, except for 1.2 and to some degree 2.0, the growth rate merely drops and does not become very small, and it is higher for later versions.

The alternations between development and release activity echo early models of large system development. In these models, it was suggested that successive releases emphasize either “progressive” or “anti-regressive” work [37, 72]. Progressive work was the development of new features, while anti-regressive work included work to improve the structural design of the software and updating documentation. Interestingly, it was shown (based on a high-level assumed mathematical model) that the best progress would be achieved by alternating these activities, rather than trying to carry them out concurrently [72]. Such alternations also occur in the simulations of Cook et al. [12], due to the assumption that activities such as refactoring absorb developer resources that would otherwise be devoted to producing new features. In our model, this alternation appears around each release of a new production version, rather than at successive releases. In principle, it can also occur multiple times within a release cycle. For example, evidence from Microsoft mentions code stabilization efforts that are performed every few months [15].

4.2. Compressed Release Model

The idealized release model discussed above hinges upon the decision to release. In the major Linux production versions these decisions were related to the development of a major feature, such

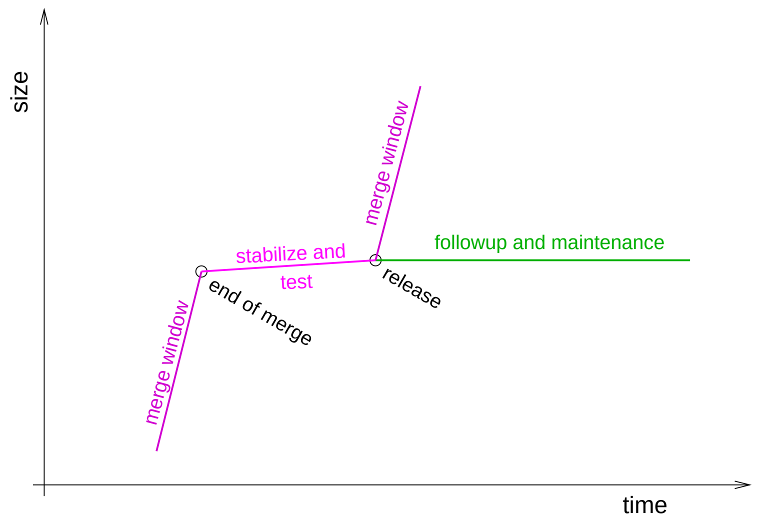

Figure 8: *Model of the anatomy of a compressed release.*

as multi-platform support and SMP support. However, such developments took an order of two years to complete, causing major delays in the release of other lesser features. With version 2.6 it was therefore decided to switch to a periodic scheme with a new release around every 2–3 months. This also reduced the pressure on developers, because if one release is missed the next one is not far off.

A faster release cycle was problematic with the idealized release model, as an underlying notion in that model is that the same people alternate between development and release activities. With the growth of Linux this became increasingly untrue. Core developers became more involved with administrating the kernel versions, while other developers more commonly contributed complete subsystems that had been developed externally. At the same time, a "stable team" was formed to take care of updates to previous production versions. This enabled a three-way parallelization of activities:

- Development of new functionality. This is done independently by many developers in their own environment (importantly, this is explicitly supported by the git model of distributed version control, where each developer has his own private copy of the codebase). Such development may take a long time, and is not reflected in any way in the Linux kernel releases and updates until it is merged during a convenient merge window.
- Stabilization and testing. This is orchestrated by the core developers, notably Linus Torvalds, in cooperation with the developers who made contributions in the most recent merge window. It is reflected as updates to the current rc version.
- Maintenance and updates of the previously released version, reflected in updates to that version.

This parallelization allows the release cycle to be compressed and completed within about 10 weeks. The compressed release cycle is shown in Fig. 8. The life of a new version starts with a

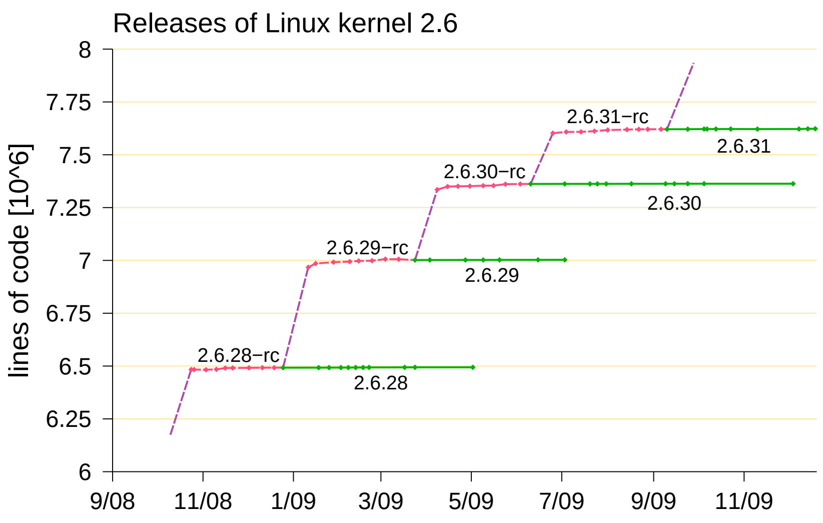

Figure 9: *Compressed releases in the Linux 2.6 series.*

merge window. This is a relatively short period (around two weeks) where developers are invited to submit their new developments[3]. When the merge window ends, a stabilization period (of around two months) takes place. The updates of this not-yet-released new version are called "release candidates" and designated by an "rc" suffix. When the new version is considered stable, it is released. In parallel, a new merge window is opened for the next version. The released version is handed over to the stable team, which performs maintenance updates (bug fixes and security patches) as needed.

Data from several recent Linux 2.6 releases is shown in Fig. 9. Note that there is no "development version", and there are no updates during the merge window. All development work is done externally by the developers in their own environment, in parallel to the release of previous versions. In few cases the first rc update serves to extend the merge window, but in most cases the rc updates do not contain any significant growth.

Given that the main changes in the compressed release model result from merging new functionality during a merge window, one may expect that the rate of growth will increase and that no code will ever be removed. This is not the case. Data collected by Greg Kroah-Hartman from the git repositories of versions 2.6.11 through 2.6.35[4] shows that thousands of lines are removed in each new version. However, additions outnumber deletions by an average factor of 2.12.

Importantly, the 2.6 releases are time-driven rather than being content-driven. Merge windows are relatively short, and the time to stabilize is also limited. If stabilization is not achieved, merged functionality may be removed and delayed to the next version. This leads to rapid dissemination

[3]These new features are expected to have been reviewed and signed off by subsystem maintainers, and incorporated in the Linux "next" version, but this is not reflected in any official releases.

[4]Available at www.kernel.org/pub/linux/kernel/people/gregkh/kernel_history/kernel_stats.ods

of innovations and developments, but also means that stable versions have a very short lifetime. As a result, some versions are singled out for "longterm" maintenance, and updated in parallel to subsequent releases.

5. The Growth of Linux

The Linux data indicates that the kernel distribution grew by a factor of 78.6 over $17\frac{1}{4}$ years — an average annual compounded growth rate of 28.8% (as measured by LoC). Portrayal of how a software system grows is a central component of the perpetual development model. "Continued growth" is also Lehman's 6th law of software evolution [43]. While not one of the original three laws proposed in 1974, it nevertheless figures prominently in the research literature. This may perhaps be attributed to the fact that it is the easiest to measure directly. For all these reasons, it is of interest to study the growth of Linux in some detail.

5.1. Measuring Growth

In Fig. 2 we used lines of code as a metric for the size of the Linux kernel. This is probably the most commonly used size metric in software engineering, and has also been used before to portray the growth of Linux [24, 64, 18]. However, there are other options. We therefore start the discussion with a comparison of different metrics. This shows that they are highly correlated, and therefore which metric is used is not very important.

The metrics we checked are the following:

- Lines of actual code. This excludes blank lines used for formatting and comment lines most commonly used in block comments. However it does include lines that contain both code and a comment.
- Total lines. This is a variant that includes all the lines: code, blank, and comments.
- The number of files. This is often used as a proxy for the number of modules. The number of modules was used by Lehman (e.g. [38]) and others to quantify the size of closed-source software, but the precise meaning of "modules" was seldom defined.
- The size of the compressed (with gzip) tar archive containing all the kernel sources, as downloaded from www.kernel.org. This metric was also used by Godfrey and Tu [24].

The first three metrics were calculated using cloc (available from cloc.sourceforge.net). This was used with all default settings, including the check for duplicate files and avoiding double-counting when this happens (surprisingly, it does). Notably, cloc counts the lines of all programming languages it can identify. In Linux this is mostly *C* (.c and .h files), assembly, and *make*, but there are also a few files in other languages. To give a notion of the distribution, the output of cloc on kernel version 2.6.39.1 is shown in Table 4. While one may claim that XML and HTML are not really programming languages and should therefore be excluded, we note that such languages contribute less than 1% of the total. We therefore believe that using the default settings does not lead to any significant errors.

Language	files	blank	comment	code
C	16087	1501193	1531754	7742940
C/C++ Header	13589	314821	536253	1632046
Assembly	1217	39850	49723	247005
XML	139	3119	948	40974
make	1390	6015	6374	22643
Perl	41	2973	2462	13900
Bourne Shell	61	638	1475	3644
yacc	5	453	322	2987
Python	18	542	267	2535
C++	1	209	57	1521
lex	5	203	237	1317
awk	8	90	79	714
Bourne Again Shell	28	74	55	446
HTML	2	58	0	378
NAnt scripts	1	87	0	356
Lisp	1	63	0	218
ASP	1	33	0	137
XSLT	6	13	27	70
sed	1	0	3	30
vim script	1	3	12	27
SUM:	32602	1870437	2130048	9713888

Table 4: *Results of counting lines of kernel version 2.6.39.1 using* cloc.

	tot. lines	*files*	*tar size*
code lines	0.99998	0.99833	0.99940
total lines		0.99828	0.99951
files			0.99823

Table 5: *Correlation coefficients between the different size metrics shown in Fig. 10.*

We noted the compressed tar archive size and applied cloc to all 1322 available versions of the Linux kernel from version 1.0 to version 2.6.39.1 (including 2.6 release candidates). As shown in Fig. 10, the different metrics are obviously very closely related. Similar-looking graphs are also obtained for other size metrics, such as the Halstead Volume or the number of functions [30]. The correspondence between the metrics is demonstrated quantitatively by calculating their correlation coefficients. The results, shown in Table 5, are all extremely close to 1. Growth as measured by files and LoC was also compared by Herraiz et al. with similar results to ours [28]. In the sequel we therefore use LoC.

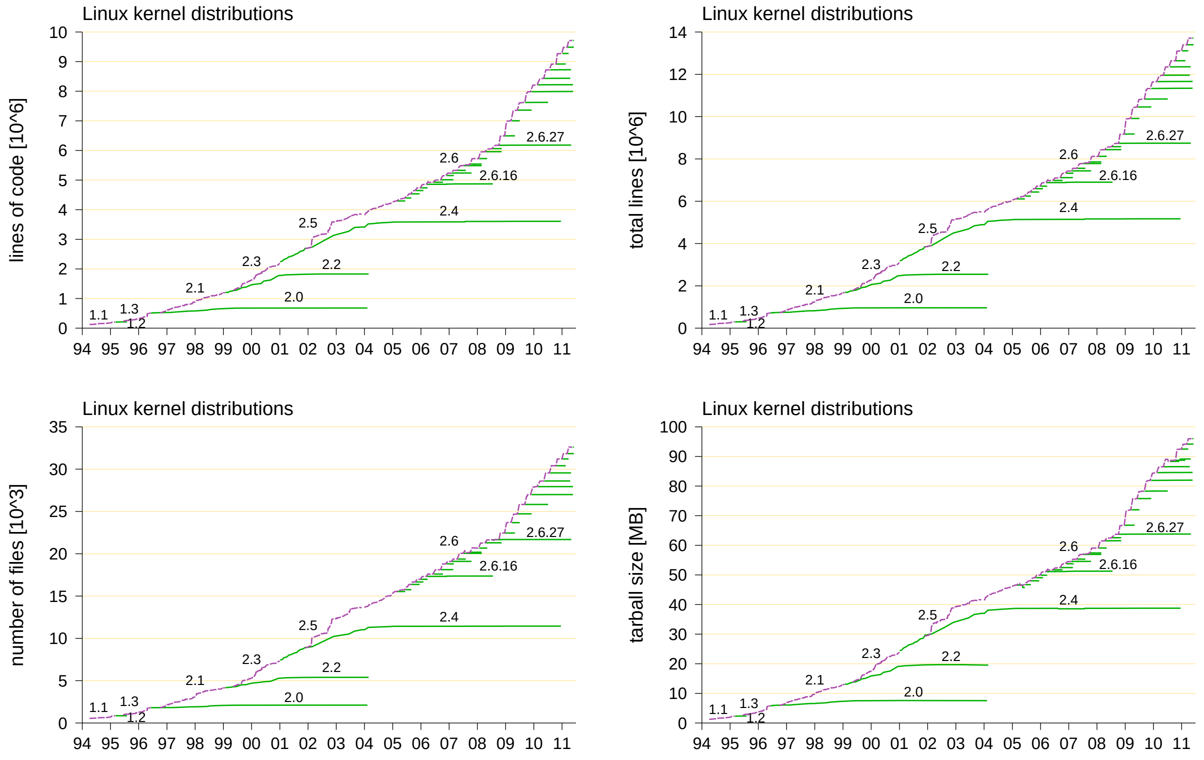

Figure 10: *Growth of Linux as reflected by different size metrics.*

5.2. Modeling Growth

The subject of the rate of growth of large-scale projects has aroused some interest in the literature. Depending on the system studied and the methodology used, researchers have concluded that growth may be sub-linear, linear, or super-linear.

Lehman and Turski, in their analysis of closed-source projects, hypothesize that the rate of growth should decrease as a result of the increasing complexity of the project [37, 44, 19]. In particular, Turski argues that the if the size of version i is s_i, and the size of version $i+1$ is s_{i+1}, then the expected relationship is [68]

$$s_{i+1} = s_i + \frac{\bar{E}}{s_i^2}$$

where $\bar{E}$ is the average effort invested in each release (Lehman's fourth law states that the rate at which effort is expended is constant, so with regularly spaced releases the effort per release should be constant as well). The inverse square increments are justified by the notion that the number of possible interactions between s_i modules is s_i^2, and the effort is spent considering all these interactions. This model leads to sublinear growth, and specifically to [61]

$$s_i \propto \sqrt[3]{i}$$

Indeed, some of Lehman's data seems to fit such a sub-linear model better than a linear model [68, 41, 40, 44]. However, other data fits a linear model quite well [42], and a linear model was also proposed by Capiluppi for several open-source projects [7, 11]. This leads to the conclusion that both models may be appropriate in certain conditions. To further complicate the issue, one of Lehman's data sets exhibited two phases of declining growth, but a significant jump in size between them.

An important consideration when studying growth rates is the independent variable. Lehman consistently used the serial numbers of successive releases, regardless of the calendarial time that passed between them [44]. This may be problematic when minor releases of different versions are interleaved, e.g. because a minor release of an old version (say release 2E) is made after the initial release of the next version (say version 3). In addition, the release rate may be highly variable, again leading to inconsistencies when using serial numbers [66]. Most researchers nowadays therefore prefer to use calendar time [24, 56].

When using calendar time the observed growth rates also differ. Paulson et al. claim that linear growth provides a good model [56], but this could be partly the result of using relatively short observation intervals of up to about a year and a half. Izurieta and Bieman also claim a linear growth rate for both FreeBSD and Linux [31].

Interestingly, several researchers have also observed super-linear growth rates — especially in the context of open-source projects, and specifically, Linux. Perhaps the first were Godfrey and Tu, who suggested a quadratic model as providing the best fit for Linux development versions [24]. Robles et al. confirm this five years later, but contend that a linear model is sufficient for BSD and 18 other projects [61]. Mens et al. studied several different metrics for the size of Eclipse, and found that four of them grew linearly, and another two quadratically [50]. Herraiz et al. compared the use of LoC with number of files, and showed that they lead to the same conclusions — including superlinear growth for many projects (but linear or even sub-linear for others) [28]. Koch extended the scope by studying thousands of projects hosted on SourceForge, and concludes that a quadratic model tends to provide better fits for most projects, especially large ones [33]. Thus we can say that the quadratic model seems to provide a good description of growth in some cases, despite having no theory regarding why growth should be quadratic.

Given that several years have passed since these previous studies were conducted, we now have much more data at our disposal. Our data in Fig. 2 clearly shows that Linux's growth rate now is even higher than what it was before. While this growth has some irregularities, it is interesting to check whether the quadratic model still provides the best fit.

However, one needs to be careful about using all the data. In particular, we suggest that the most representative results will be obtained by using a carefully chosen subset. The problem is that curve-fitting techniques take all the data points into consideration, and typically try to minimize the (squared) deviation between the points and the model. If many points are concentrated at some location (in our case, due to an abnormally high rate of updates), this tends to force the model to pass through that location. But we are more interested in the general trend over time. We therefore chose to sample the data at monthly granularity. Thus we use the first development update in each month, or, in those cases where a development version does not exist, the first production update.

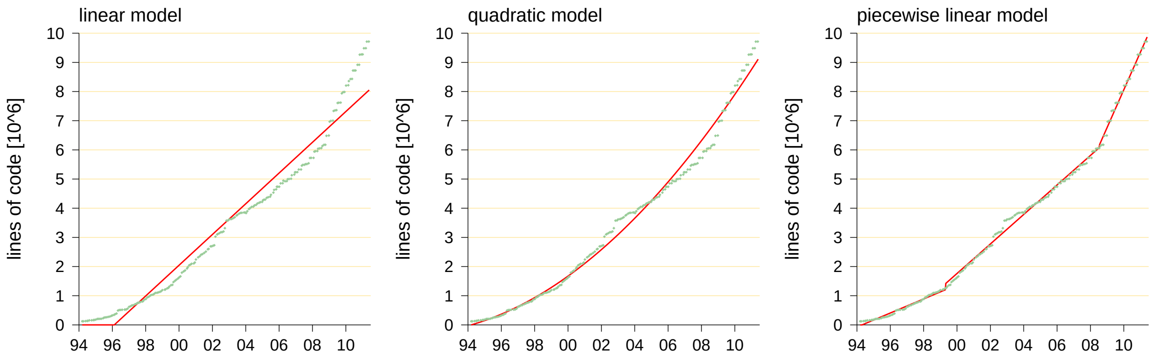

Figure 11: *Models of the Linux growth rate.*

model	*total error*	*average error*
linear	81,116,099	395,688
quadratic	42,017,782	204,965
cubic	64,497,025	314,619
exponential	75,262,951	367,136
quadratic-exponential	27,265,871	133,004
piecewise linear	17,271,571	84,252

Table 6: *Errors of models of Linux growth, in lines of code. Note that the scale is 10 million, so an average error of 100,000 is only 1%. Also, the growth in 2.6 has a step shape, leading to unavoidable error by any smooth model.*

In the 2.6 series we use the first release of each new version, and the first rc update in intervening months. This leads to using a total of 205 data points.

The results of fitting various models to this data are shown in Fig. 11. In the linear models we imposed a lower-bound of 0 in cases where the model suggested a negative size. Obviously the linear model is not a very good fit, because the Linux growth rate is increasing with time. We nevertheless note that the correlation coefficient of the linear model with the data is pretty high, at 0.978. The quadratic model is better, and provides a reasonable fit up to about 2005. However, it is less satisfying after that. We also checked a cubic model and an exponential model, but they were inferior to the quadratic model.

When looking at the data, it is appealing to consider it as being composed of two phases: from the beginning to the two large jumps in version 2.5 towards the end of 2002, and from 2003 to the end. Trying to fit each of these phases independently leads to a good fit with a quadratic model for the first phase (thereby reconfirming the previous results of Godfrey and Tu and others), and a reasonable fit for an exponential model for the second phase. This two-phase model achieves much lower error than the previous models (Table 6).

An even better fit is obtained by a simple three-segment piecewise linear model. This model dissects the timeline of Linux development into three phases: from the initial release to the 2.2 version in May 1999, from the beginning of 2.3 to 2.6.26 in July 2008, and since 2.6.27rc in

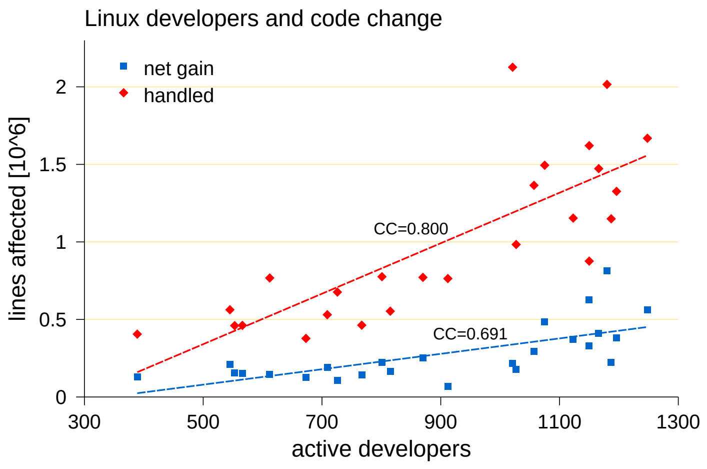

Figure 12: *Correlation between number of developers and change in code size. Data by Greg Kroah-Hartman for kernels 2.6.11 through 2.6.35.*

August 2008 to the end in mid 2011. In these segments the linear fits imply average growth rates of 660 lines per day, 1385 lines per day, and 3541 lines per day respectively. The three segments exhibit correlation coefficients of 0.987, 0.996, and 0.993, and the combined model achieves a significantly lower error than any of the other models checked (this could possibly be improved further by using a fourth initial segment, as version 1.1 hardly showed any growth). In any case, the three-phase linear model provides the best description of the growth of Linux so far, and reconciles the linear growth models with the increasing growth rate. However, it does not make predictions about how long phases will be and in general what may happen in the future.

Regardless of the exact growth rate model, it is obvious that Linux is now growing faster than ever. An interesting question is how such an increasing growth rate is sustained. Capiluppi at al. found that a reduced growth rate of an agile project was correlated with a restructuring of the company and a big reduction in the size of the programming team [8]. In the case of Linux it is not unreasonable to assume that the opposite is happening: as more developers join the ranks [13], the total rate of growth increases. Fig. 12 shows supportive evidence. This is data collected by Greg Kroah-Hartman from the Linux git repository, and specifically versions 2.6.11 through 2.6.35[5]. The data collected includes the number of developers that perform commits on each version, as well as the number of lines added, deleted, and modified. We use this to calculate the net gain of lines in each version (by subtracting the deleted line count from the added line count) and the total lines handled in each version (the sum of the three counts). As the figure shows, there is a good correlation between the number of active developers and the net gain, and an even better correlation with the number of handled lines. Similar results (with an even higher correlation) were found by Koch and Schneider for the GNOME project [34].

[5]Available at www.kernel.org/pub/linux/kernel/people/gregkh/kernel_history/kernel_stats.ods

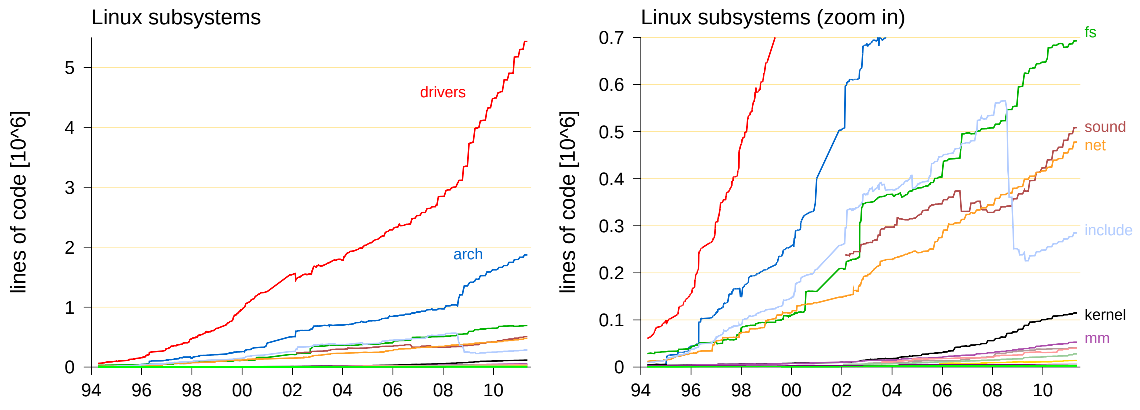

Figure 13: *Growth of Linux subsystems (top-level directories). The unmarked small ones are (in order of final size) crypto, security, lib, block, ipc, virt, init, firmware, and usr.*

Based on the above observation, we may speculate that the increased growth rate of Linux and other open-source software projects is a manifestation of a positive feedback effect. This is the combination of two mutually-reinforcing processes. The first is that a successful and useful project attracts more highly-motivated and capable developers [10]. The second is that the aggregation of such developers propels the project to ever greater heights. This is expected to be possible in open-source projects but not in closed-source projects, because of the much more flexible and budget-free staffing of open-source projects. We contend that this positive feedback generates super-linear growth, and that this dominates the possible detrimental effects of increased complexity.

An interesting related observation is that the superlinear growth and concurrent growth in number of developers seem to contradict Brooks's Law [6]: instead of making progress slower, the added developers actually make it faster. The implication is that the required communications between developers do not grow as much as was suggested by Brooks. This phenomenon, which has been observed before [1], may be attributed to the modular structure of the system and to efficient mechanisms for disseminating and recording design information such as the Linux kernel mailing list.

5.3. Growth of Subsystems

The Linux kernel is actually composed of many largely independent subsystems. For example, even within the core of the kernel, features like file systems, memory management, and communications are implemented as distinct subsystems. Then there are the multitude of drivers, and interfaces to many different architectures. It is therefore interesting to check how each such subsystem grows, and whether they all contribute equally to the overall growth of Linux.

Results of measuring the subsystems (as reflected by top-level directories) are shown in Fig. 13. In general most subsystems grow similarly to the whole system, implying that they all grow at roughly the same rate relative to their respective sizes. To quantify this, we calculate the correlation coefficients of the subsystem sizes with the total system size. For each subsystem, this is done over

	CC with	lin. reg. of %	
subsystem	*total*	*slope*	R^2
drivers	0.99564	8.85E-6	1.7E-6
arch	0.99669	0.00328	0.262
fs	0.99021	-0.00282	0.259
sound	0.95417	-0.00343	0.866
net	0.99148	-0.00234	0.717
include	0.71278	-0.00329	0.489
kernel	0.94324	-0.00026	0.054
mm	0.98060	-0.00047	0.438
crypto	0.96780	0.00037	0.814
security	0.94663	0.00037	0.579
lib	0.98402	4.41E-5	0.094
block	0.96162	-3.38E-5	0.274
ipc	0.94322	-0.00035	0.539
virt	0.97420	5.14E-5	0.687
init	0.90513	-0.00011	0.652
firmware	0.68636	-3.21E-5	0.847
usr	0.75877	1.40E-6	0.032

Table 7: *Correlations of subsystem sizes with the full system size (CC), and linear regression of the percentage of each subsystem from the total system size.*

the range of versions where this subsystem exists, using only development (and rc) versions. The results are shown in Table 7, and indicate a very strong correlation in most cases.

To augment this data, we also calculate the percent of the total Linux code that belongs to each subsystem. Using this metric, proportional growth of all subsystems should lead to perfectly horizontal lines [24]. The actual results are shown in Fig. 14, indicating some fluctuations and discrete jumps. Calculating the linear regression of these percentages leads to very small slopes (Table 7, given in percentage points change per year). However, the R^2 values indicate that linear models are in some cases inappropriate for this data.

The largest subsystem by far, and thus also the one that is growing at the fastest rate, is drivers. The second largest is arch (architecture support). These have been recognized as the largest many times in the past, and together represent about 70% of the kernel [24, 30]. Due to their dominance, it is not surprising that these two subsystems have the highest correlations with the total system size. The two smallest and most recent subsystems, firmware and usr, exhibit relatively low correlations, but due to their small size this is not very meaningful.

According to Fig. 14 the drivers subsystem is actually growing relative to all others. However, it suffered a large drop in relative size in version 2.5.5 on 20 Feb 2002. This was the result of moving the subdirectory drivers/sound/ to the top level and creating the sound subsystem. This move affected over 230,000 lines of code.

The arch subsystem, in contradistinction, exhibits a moderate reduction of relative size with

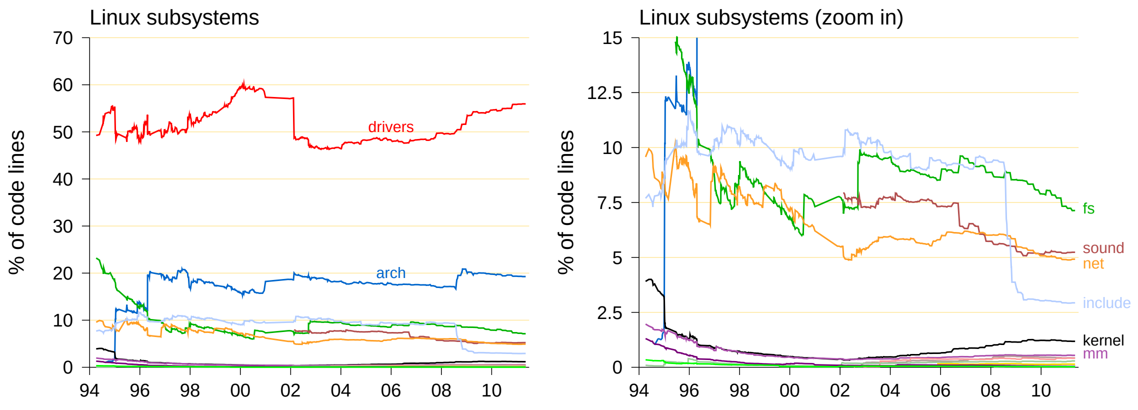

Figure 14: *Relative sizes of Linux subsystems as a percentage of the total code size.*

time. However, it exhibits several discrete upwards jumps that counteract this downwards trend. The most significant of these occur in version 1.1.78 on 9 January 1995 and in version 1.3.94 on 22 April 1996. The first is largely due to adding math emulation support in the Intel i386 architecture. The second is largely due to added support for the Motorola 68000 architecture. A similar gradual reduction of relative size with occasional upwards jumps is also observed for subsystems fs and net.

Another subsystem that exhibits an interesting pattern is include, which had a huge drop in size in mid 2008 (as may be expected, this abnormal behavior leads to a relatively low correlation with the total system size). This actually took place over a couple of months, in successive updates of version 2.6.27-rc. The changes are apparently moving several subdirectories of assembler-related .h files (e.g. include/asm-alpha/) to the arch subsystem (e.g. /arch/alpha/include/asm/). And indeed, a corresponding increase in arch can be seen in Figs. 13 and 14.

The conclusion from the above observations is that evolution is not always a smooth process. In many cases it is punctuated by relatively large changes in a short period of time (as has been observed previously by Godfrey and Tu [24] and by Cook et al. [12]). These can be the addition of a large body of code, possibly developed externally, or a reorganization where code is moved from one place to another. However, when the complete system is composed of many subsystems, the overall effect may appear smoother and more consistent than the dynamics of each individual subsystem.

6. Possible Implications of the Model

The continuous nature of perpetual development has many implications regarding how software is developed. Here we speculate about some of them, and the applicability beyond the Linux case study. Note that most of this discussion is not yet supported by data, and should therefore be regarded as proposing promising avenues for further research.

The Question of Architecture. A major vulnerability of many software development processes is the definition of the system's architecture at a relatively early stage. This is a crucial and singular point — the analysis and design are geared towards the definition of the architecture, and the architecture is the basis for the whole implementation. The problem becomes much more acute under perpetual development, as it is explicitly acknowledged that the initial design will need to accommodate unforeseen additions for many years to come.

A possible way to alleviate the problem is to use an architecture that is inherently open and flexible [21]. In particular, architectures that support continuing change include the following:

- A "kernel-based" two-tier system, with a stable core and a dynamic set of libraries where things can change quickly and relatively independently. Examples include emacs with its basic core and many user-developed Lisp modules, and matlab with its extensive functional libraries [60].
- A multi-tier service-oriented architecture, e.g. the agglomeration of Internet services used in large e-commerce sites such as amazon.com. This is essentially an open system architecture based on independent components, with small well-defined interfaces [14]. At any given time new components can be introduced, or existing components may be redone or extended, with little effect on the others.
- An explicit component-based architecture, where the desired functionality is obtained by agglomerating multiple independent components. An example is the Eclipse integrated development environment with its many plugins [70].

In Linux such compartmentalization is used to a certain degree. For example, file systems are largely independent of memory management, and drivers (which constitute a large fraction of the code) are all essentially independent of each other. Thus components can each be modified independently with little if any effect on other components.

The parallel development that is inherent in the perpetual development model may also help to cope with radical changes to the architecture. A project may branch off an exploratory architecture development branch, that is developed in parallel and independently from the main development backbone. If the new architecture succeeds it will eventually be adopted, and other modules will be ported to adhere to it. This is analogous to redundancy in biological systems, which allows many mutations with no deleterious effects on the organism, at the same time immediately benefiting from advantageous mutations.

The Problem of Conservation of Familiarity. One of Lehman's laws of software evolution is the conservation of familiarity — that the rate of evolution is constrained by what users and developers can absorb [39]. Perpetual development thus requires tools and approaches that facilitate conservation of familiarity.

A new developer that enters an existing project suffers an inherent disadvantage: he or she sees the product of a potentially long evolutionary process, that does not reflect a clean and coherent design because there never was one. There is no distinction between original basic functions and later add-ons to fix or support emerging issues. Thus understanding an existing project all at once is much harder than following its progression as it is being done.

A possible solution for getting up to speed with existing code is a code browser which depicts the edit history. For example, "history flows" enable one to see when each part of the code originated, and who added it [69]. Such a view could be synchronized with a code viewer, that uses a version control system to reconstruct the code as it was at select earlier times. This enables the exclusion of all subsequent additions and modifications, and allows a new developer to easily recreate and follow the process that led to the code as it now exists. Synchronizing across multiple files (e.g. code and header files) will enable to highlight all the code that was committed within a specified time period, and thus represents the addition of a certain set of features.

A similar problem happens for users: early adopters have an inherent advantage, as they initially learn to use the very basic features that are implemented first, and then get to learn the more advanced and involved features at a slow pace as they are implemented. New users, on the other hand, have to assimilate the whole caboodle in one big lump. A possible way to help them out is to pre-define several views with increasing sophistication. In particular, it is important to have a novice version with limited features to get started and learn to use the system, that stays consistent across new releases. In Linux, the importance of conservation of familiarity is witnessed by the fact that users may refrain from adopting newer versions of the system, thereby necessitating the continued maintenance of older versions. But support for backward compatibility may be a better alternative, as users will be able to benefit from improved internal implementations, and it may reduce the burden of maintaining previous versions separately.

Maintaining Institutional Knowledge. Related to the above two points is the issue of maintaining knowledge. Classical software development processes, such as a waterfall model, are document-heavy [62]. Documents serve as milestones, and as evidence that a phase has been completed successfully and the next phase should start. Parnas has suggested that even in evolutionary scenarios, where documentation cannot be completed in advance, it is worthwhile to "fake it" and maintain documentation that is in sync with the software [55]. For example, this records important design decisions and enables new developers to better understand the project.

However, in projects like Linux it seems that detailed documentation is not maintained, which raises the question of how they manage to continue to grow and flourish. This question is particularly poignant when one takes into account the growth of Linux and the constant addition of new developers — a situation that might lead to problems of learning and communication as suggested by Brooks's Law [6]. The answer seems to be that alternative mechanisms have been created to store collective knowledge about the project [20]. These are mainly bug repositories and mailing lists — in the case of Linux, the Linux kernel mailing list, where practically all technical discussions occur in real time. In addition, large amounts of knowledge may actually be stored only in the heads of lead developers. This may be less risky than it seems, as lead open-source developers typically maintain close long-term relationships with their projects, irrespective of their current employment status.

The Curse of Successful Maintenance. The field of software engineering grew from the perception that the practice of software development is in crisis. Too many projects are late, over budget, or do not provide the expected functionality. This is especially problematic with large-scale systems, where hundreds of millions of dollars may be wasted on failed efforts. Many such failures may be rooted in a failure to recognize and use perpetual development as described above [21, 73, 17]. Basically, projects that attempt to do too much at once will most probably fail in one way or another [29]. An ironic outcome of this is that the burst of the hi-tech bubble in 2001 led to an *improvement* in project success rates, because the reduced budgets led to smaller, more focused projects [27]. This underscores the importance of the incremental approach to project development: using increments reduces the scope being considered at each step, which makes it realizable.

An important class of oversized projects is those that aim to replace a previous system that has become outdated. Famous examples of this type are the FAA air traffic control system automation project of 1984–1994 [57], and the more recent FBI virtual case file [25]. The problem in these cases is that the old system was used successfully for a long time. Placing the focus on maintaining it rather than on continued development — and, ironically, being successful in such maintenance — led to an ever-growing gap between the system capabilities and what was really needed. Then, when an upgrade was attempted, it was too late: the gap was too large to bridge, and trying to do so in one fell swoop failed.

Some projects are naturally perpetual, e.g. operating systems. The question of stopping doesn't come up at all — it is clear that development of new versions will continue ceaselessly. The problem is that projects like the FAA air traffic control system and the FBI virtual case file look like one-shot affairs, that should be designed, implemented, and installed in a single phase. But this is not necessarily so. Applying the principles of perpetual development, and seeking constant improvements to the system in terms of both the implementation and the hardware base, may save them from subsequent failure [17].

Towards Programming as a Service. Finally, the notion of perpetual development has an important impact on how software development is funded and contracted. In particular, an important consideration for organizations contracting software systems is the need to include evolution and longevity in the contract framework [47]. Support for evolution means the contractor will continue to develop and adapt the software as needed. This may be achieved by dissecting the project into small increments, and extending the contract incrementally to follow the progress. Support for longevity means that if the contractor is unable or unwilling to continue work on the project, he will allow others to do so, e.g. by providing access to source code. Much of the problems with maintenance of legacy systems stem from lack of such facilities (although lack of documentation, rigid architectures, and so on should not be underestimated as well).

Conventional software contracting includes an inception phase where the feasibility of the project is verified, and an elaboration phase where the details are fleshed out, leading to the signing of a contract. Such a contractual framework becomes increasingly less relevant as the length of the project is extended and the features that were identified at the outset become a smaller fraction of the whole effort. In fact, additional development — and not maintenance — is the service that should be contracted throughout the lifetime of the project. An interesting future research question is how to define and quantify the effort and outcomes of a software development effort, in a way that can be used in lieu of a contract that specifies the anticipated developed product.

7. Conclusions and Future Work

The notion of perpetual development is not new. It has been used naturally and successfully by many large-scale projects, especially but not exclusively in the open-source community. However, it seems to be badly underrepresented in the professional literature and especially in software engineering textbooks, which focus on isolated instances of development rather than on the continuation between many successive developments. It is also not prominent in the literature about open-source and agile development, which tends to focus on managerial and process issues (rapid releases, incremental setting of targets, pair programming, etc.).

The perpetual development model helps elucidate several aspects of large-scale evolutionary system development. To summarize, the main contributions of this model are the following.

- Identifying growth (as in Lehman's 6th law) as the central element in software evolution, at least for software systems like Linux that correspond to this model. The continued growth is possibly enabled by positive feedback between the system and a growing developer base.
- Stipulating that multiple versions co-exist at the same time (as was also specified in the model of Rajlich and Bennett [59]). This separates evolution, which happens in the main development branch, from maintenance, which happens in production branches.
- Recognizing release points as the points where new branches are forked, and formulating the release process with its stabilization and followup phases and possible variants.

At the same time, the model points out directions that would benefit from additional research. These include

- Linux is a single system. An important goal for additional work is therefore to check the degree to which the model developed here applies to other large systems. Initial supportive evidence can be collected from software distribution sites. For example, the Eclipse site indicates that each new version is preceded by a series of "milestone" releases (equivalent to our development updates), followed by a series of release candidates, and that all this happens in parallel to the (typically very few) maintenance updates of the previous version. A figure similar to our Fig. 2 portraying the evolution of BIND (a DNS server project) is given by Xie et al. [74]. Indeed, even some of Lehman's early data shows overlap between development and maintenance versions, where growth is observed mainly in new releases (the FW dataset in [44]).
- Evolution by perpetual development leads to continued addition of functionality rather than change of functionality. An interesting future research question is to contrast these two aspects of evolution. In particular, is there evolution that manifests itself as change without growth (e.g. in the context of software product lines)? Alternatively, does the growth actually reflect a shift in focus, where part of the codebase falls out of use but is not removed?
- In perpetual development evolution and maintenance both happen after delivery, but in different contexts. We also saw that they may be interdependent, e.g. in the initial years of Linux versions 2.2 and 2.4. An interesting future research question is the characterization

and analysis of such interactions, including the degree that new developments are propagated into current production versions, and how does maintenance of existing versions influence the development of subsequent versions.

- The Linux case study shows that considerable work is expended on preparing code for release, as reflected by a sequence of release candidates. Does this have measurable effects on properties of the code? And what are the inputs that lead to the decision that the new release is indeed stable enough for release?
- Finally, perpetuity in software development defies intuition. It seems incredible that a system can grow and grow without bounds. But Linux has been growing at a superlinear rate for 17 years, and does not show signs of slowing down despite its large current size. While a large part of this is due to drivers and architecture support, the numbers are still very impressive even if these are excluded. An important challenge is therefore to understand whether this is due to misconceptions about the detrimental effects of growth, or to active work (and if so, what precisely) to counter such effects.

Acknowledgment

This work was supported by a grant from The Wolfson Fund.

Appendix: Linux Versions

The location of the main Linux versions is self evident on www.kernel.org. However, this is not always the case for versions that are part of the release process, including pre-release versions, testing versions, and release candidates. Table 8 lists where these can be found. Note that all kernel versions are stored under /pub/linux/kernel/, so a location of "v1.0/linux-1.0" actually means the URL www.kernel.org/pub/linux/kernel/v1.0/linux-1.0.tar.gz.

References

[1] P. J. Adams, A. Capiluppi, and C. Boldyreff, "*Coordination and productivity issues in free software: The role of Brooks' law*". In *Intl. Conf. Softw. Maintenance*, pp. 319–328, Sep 2009.

[2] G. Antoniol, U. Yillano, E. Merlo, and M. Di Penta, "*Analyzing cloning evolution in the Linux kernel*". *Inf. & Softw. Tech.* **44(13)**, pp. 755–765, Oct 2002.

[3] K. Beck, "*Embracing change with extreme programming*". *Computer* **32(10)**, pp. 70–77, Oct 1999.

[4] B. Boehm, "*Making a difference in the software century*". *Computer* **41(3)**, pp. 32–38, Mar 2008.

[5] B. W. Boehm, "*A spiral model of software development and enhancement*". *Computer* **21(5)**, pp. 61–72, May 1988.

[6] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

series	*location*	*comments*
initial	Historic/linux-0.01 Historic/old-versions/linux-0.* Historic/v0.99/linux-0.99.*	pre 1.0 versions not used in our study numbering is erratic
1.0	v1.0/linux-1.0	only one version
1.1	v1.1/v1.1.0 v1.1/linux-1.1.*	 many serials missing
1.2	v1.2/linux-1.2.*	
1.3	v1.3/linux-1.3.* v1.3/linux-pre2.0.*	1.3.1 is missing
2.0	v2.0/linux-2.0.*	
2.1	v2.1/linux-2.1.* v2.1/linux-2.2.0-pre*	
2.2	v2.2/linux-2.2.*	
2.3	v2.3/linux-2.3.* v2.3/linux-2.3.99-pre* v2.4/old-test-kernels/linux-2.4.0-*	 note location under v2.4
2.4	v2.4/linux-2.4.*	2.4.11 is marked “dontuse” but we use it
2.5	v2.5/linux-2.5.* v2.6/pre-releases/linux-2.6.0-test*	 note location under v2.6
2.6	v2.6/linux-2.6.* v2.6/testing/v2.6.*/linux-2.6.*-rc* v2.6/longterm/v2.6.*/linux-2.6.*	 release candidates additional longterm versions

Table 8: *Location of Linux versions at* www.kernel.org.

[7] A. Capiluppi, “*Models for the evolution of OS projects*”. In *Intl. Conf. Softw. Maintenance*, pp. 65–74, Sep 2003.

[8] 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 29th *Intl. Conf. Softw. Eng.*, pp. 511–518, May 2007.

[9] A. Capiluppi, J. M. González-Barahona, I. Herraiz, and G. Robles, “*Adapting the “staged model of software evolution” to free/libre/open source software*”. In 9th *Intl. Workshop Principles of Softw. Evolution*, pp. 79–82, Sep 2007.

[10] A. Capiluppi and M. Michlmayr, “*From the cathedral to the bazaar: An empirical study of the lifecycle of volunteer community projects*”. In *Open Source Development, Adoption, and Innovation*, pp. 31–44, Jun 2007. (IFIP vol. 234).

[11] A. Capiluppi, M. Morisio, and J. F. Ramil, “*Structural evolution of an open source system: A case study*”. In 12th *IEEE Intl. Workshop Program Comprehension*, pp. 172–182, Jun 2004.

[12] S. Cook, R. Harrison, and P. Wernick, “*A simulation model of self-organising evolvability in software systems*”. In *IEEE Intl. Workshop Software Evolvability*, pp. 17–22, Sep 2005.

[13] J. Corbet, G. Kroah-Hartman, and A. McPherson, "*Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it*". URL www.linuxfoundation.org/docs/lf_linux_kernel_development_2010.pdf, Dec 2010. (Downloaded 9 Jun 2011).

[14] F. Cuadrado, B. García, J. C. Dueñas, and H. A. Parada, "*A case study on software evolution towards service-oriented architecture*". In 22nd *Intl. Conf. Advanced Inf. Netw. & App. – Workshops*, pp. 1399–1404, Mar 2008.

[15] M. A. Cusumano and R. W. Selby, "*How Microsoft builds software*". *Comm. ACM* **40(6)**, pp. 53–61, Jun 1997.

[16] M. Denne and J. Cleland-Huang, "*The incremental funding method: Data-driven software development*". *IEEE Softw.* **21(3)**, pp. 39–47, May/Jun 2004.

[17] P. J. Denning, C. Gunderson, and R. Hayes-Roth, "*Evolutionary system development*". *Comm. ACM* **51(12)**, pp. 29–31, Dec 2008.

[18] A. Deshpande and D. Riehle, "*The total growth of open source*". In 4th *Conf. Open Source systems*, Sep 2008.

[19] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "*Does code decay? assessing the evidence from change management data*". *IEEE Trans. Softw. Eng.* **27(1)**, pp. 1–12, Jan 2001.

[20] L. Gasser, W. Scacchi, G. Ripoche, and B. Penne, "*Understanding continuous design in F/OSS projects*". In 16th *Intl. Conf. Softw. & Syst. Eng. & Apps.*, Dec 2003.

[21] T. Gilb, *Principles of Software Engineering Management*. Addison-Wesley, 1988.

[22] M. W. Godfrey and D. M. German, "*The past, present, and future of software evolution*". In 24th *Intl. Conf. Softw. Maintenance*, pp. 129–138, Sep 2008. (Special track on Frontiers of Software Maintenance).

[23] M. W. Godfrey, D. Svetinovic, and Q. Tu, "*Evolution, growth, and cloning in Linux: A case study*". In *CASCON workshop on Detecting Duplicated and Near Duplicated Structures in Large Software Systems: Methods and Applications*, 2000. (Presentation available at plg.uwaterloo.ca/˜migod/papers/2000/cascon00-linuxcloning.pdf).

[24] M. W. Godfrey and Q. Tu, "*Evolution in open source software: A case study*". In 16th *Intl. Conf. Softw. Maintenance*, pp. 131–142, Oct 2000.

[25] H. Goldstein, "*Who killed the virtual case file?*" *IEEE Spectrum* **42(9INT)**, pp. 18–29, Sep 2005.

[26] D. Harel, "*Statecharts in the making: A personal account*". *Comm. ACM* **52(3)**, pp. 67–75, Mar 2009.

[27] F. Hayes, "*Chaos is back*". *Computerworld* Nov 2004. URL http://www.computerworld.com/s/article/97283/Chaos_Is_Back.

[28] I. Herraiz, G. Robles, J. M. González-Barahona, A. Capiluppi, and J. F. Ramil, "*Comparison between SLOCs and number of files as size metrics for software evolution analysis*". In 10th *Conf. Softw. Maintenance & Reengineering*, pp. 206–213, Mar 2006.

[29] C. A. R. Hoare, "*The emperor's old clothes*". *Comm. ACM* **24(2)**, pp. 75–83, Feb 1981.

[30] A. Israeli and D. G. Feitelson, "*The Linux kernel as a case study in software evolution*". *J. Syst. & Softw.* **83(3)**, pp. 485–501, Mar 2010.

[31] C. Izurieta and J. Bieman, "*The evolution of FreeBSD and Linux*". In 5th *Intl. Symp. Empirical Softw. Eng.*, pp. 204–211, Sep 2006.

[32] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang, "*Towards an ontology of software maintenance*". *J. Softw. Maintenance: Res. & Pract.* **11(6)**, pp. 365–389, Nov/Dec 1999.

[33] S. Koch, "*Software evolution in open source projects—a large-scale investigation*". *J. Softw. Maintenance & Evolution: Res. & Pract.* **19(6)**, pp. 361–382, Nov/Dec 2007.

[34] S. Koch and G. Schneider, "*Results from software engineering research into open source development projects using public data*". In *Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, H. R. Hansen and W. H. Janko (eds.), Wirtschaftsuniversität Wien, 2000.

[35] P. Kruchten, "*A rational development process*". *Crosstalk* **9(7)**, pp. 11–16, Jul 1996.

[36] C. Larman and V. R. Basili, "*Iterative and incremental development: A brief history*". *Computer* **36(6)**, pp. 47–56, Jun 2003.

[37] M. M. Lehman, "*Programs, cities, students – limits to growth*". In *Programming Methodology*, D. Gries (ed.), Springer Verlag, 1978.

[38] M. M. Lehman, "*Programs, life cycles, and laws of software evolution*". *Proc. IEEE* **68(9)**, pp. 1060–1076, Sep 1980.

[39] M. M. Lehman, "*On understanding laws, evolution, and conservation in the large-program life cycle*". *J. Syst. & Softw.* **1**, pp. 213–221, 1980.

[40] M. M. Lehman, "*Laws of software evolution revisited*". In 5th *European Workshop on Software Process Technology*, pp. 108–124, Springer Verlag, Oct 1996. Lect. Notes Comput. Sci. vol. 1149.

[41] M. M. Lehman and F. N. Parr, "*Program evolution and its impact on software engineering*". In 2nd *Intl. Conf. Softw. Eng.*, pp. 350–357, Oct 1976.

[42] M. M. Lehman, D. E. Perry, and J. F. Ramil, "*Implications of evolution metrics on software maintenance*". In 14th *Intl. Conf. Softw. Maintenance*, pp. 208–217, Nov 1998.

[43] M. M. Lehman and J. F. Ramil, "*The impact of feedback in the global software process*". *J. Syst. & Softw.* **46(2-3)**, pp. 123–134, Apr 1999.

[44] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perrry, and W. M. Turski, "*Metrics and laws of software evolution – the nineties view*". In 4th *Intl. Software Metrics Symp.*, pp. 20–32, Nov 1997.

[45] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "*Characteristics of application software maintenance*". *Comm. ACM* **21(6)**, pp. 466–471, Jun 1978.

[46] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "*Analysis of the Linux kernel evolution using code clone coverage*". In 4th *Intl. Workshop Mining Softw. Repositories*, May 2007.

[47] C. M. Lott, "*Breathing new life into the waterfall model*". *IEEE Softw.* **14(5)**, pp. 103–105, Sep/Oct 1997.

[48] M. L. Maher and J. Poon, "*Modelling design exploration as co-evolution*". *Microcomputers in Civil Engineering* **11(3)**, pp. 195–209, 1996.

[49] D. D. McCracken and M. A. Jackson, "*Life cycle concept considered harmful*". *Softw. Eng. Notes* **7(2)**, pp. 29–32, Apr 1982.

[50] T. Mens, J. Fernández-Ramil, and S. Degrandsart, "*The evolution of Eclipse*". In *Intl. Conf. Softw. Maintenance*, pp. 386–395, Sep 2008.

[51] E. Merlo, M. Dagenais, P. Bachand, J. S. Sormani, S. Gradara, and G. Antoniol, "*Investigating large software system evolution: The Linux kernel*". In 26th *Comput. Softw. & Apps. Conf.*, pp. 421–426, Aug 2002.

[52] M. Michlmayr, F. Hunt, and D. Probert, "*Release management in free software projects: Practices and problems*". In *Open Source Development, Adoption, and Innovation*, pp. 295–300, Jun 2007. (IFIP vol. 234).

[53] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "*Evolution patterns of open-source software systems and communities*". In 5th *Intl. Workshop Principles of Softw. Evolution*, pp. 76–85, May 2002.

[54] Y. Padioleau, J. L. Lawall, and G. Muller, "*Understanding collateral evolution in Linux device drivers*". In *EuroSys*, pp. 59–71, Apr 2006.

[55] D. L. Parnas and P. C. Clements, "*A rational design process: How and why to fake it*". *IEEE Trans. Softw. Eng.* **SE-12(2)**, pp. 251–257, Feb 1986.

[56] J. W. Paulson, G. Succi, and A. Eberlein, "*An empirical study of open-source and closed-source software products*". *IEEE Trans. Softw. Eng.* **30(4)**, pp. 246–256, Apr 2004.

[57] T. S. Perry, "*In search of the future of air traffic control*". *IEEE Spectrum* **34(8)**, pp. 18–35, Aug 1997.

[58] V. Rajlich, "*Changing the paradigm of software engineering*". *Comm. ACM* **49(8)**, pp. 67–70, Aug 2006.

[59] V. T. Rajlich and K. H. Bennett, "*A staged model for the software life cycle*". *Computer* **33(7)**, pp. 66–71, Jul 2000.

[60] E. S. Raymond, "*The cathedral and the bazaar*". URL www.catb.org/˜esr/writings/cathedral-bazaar/cathedral-bazaar, 2000.

[61] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz, "*Evolution and growth in large libre software projects*". In 8th *Intl. Workshop Principles of Softw. Evolution*, pp. 165–174, Sep 2005.

[62] W. W. Royce, "*Managing the development of large software systems*". In *Proc. IEEE WESCON*, pp. 1–9, Aug 1970. (Reprinted in 9th *Intl. Conf. Softw. Eng.*, pp. 328–338, 1987.).

[63] S. R. Schach, T. O. S. Adeshiyan, D. Balasubramanian, G. Madl, E. P. Osses, S. Singh, K. Suwanmongkol, M. Xie, and D. G. Feitelson, "*Common coupling and pointer variables, with application to a Linux case study*". *Software Quality J.* **15(1)**, pp. 99–113, March 2007.

[64] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "*Maintainability of the Linux kernel*". *IEE Proc.-Softw.* **149(2)**, pp. 18–23, 2002.

[65] L. G. Thomas, S. R. Schach, G. Z. Heller, and J. Offutt, "*Impact of release intervals on

empirical research into software evolution, with applications to the maintainability of Linux". *IET Softw.* **3(1)**, pp. 58–66, Feb 2008.

[66] L. G. Thomas, S. R. Schach, Z. G. Heller, and J. Offutt, "*Impact of release intervals on empirical research into software evolution, with application to the maintainability of Linux*". *IET Softw.* **3(1)**, pp. 58–66, Feb 2009.

[67] A. Tomer and S. R. Schach, "*The evolution tree: A maintenance-oriented software development model*". In 4th *European Conf. Softw. Maintenance & Reengineering*, pp. 209–214, Mar 2000.

[68] W. M. Turski, "*Reference model for smooth growth of software systems*". *IEEE Trans. Softw. Eng.* **22(8)**, pp. 599–600, Aug 1996.

[69] F. B. Viégas, M. Wattenberg, and K. Dave, "*Studying cooperation and conflict between authors with history flow visualizations*". In *Conf. Human Factors in Comput. Syst.*, pp. 575–582, Apr 2004.

[70] M. Wermelinger and Y. Yu, "*Analyzing the evolution of Eclipse plugins*". In 5th *Intl. Workshop Mining Softw. Repositories*, pp. 133–136, May 2008.

[71] Wikipedia, "*Software release life cycle*". URL en.wikipedia.org/wiki/Software_release_life_cycle. (Visited 25 Dec 2010).

[72] C. M. Woodside, "*A mathematical model for the evolution of software*". *J. Syst. & Softw.* **1(4)**, pp. 337–345, 1980.

[73] S. Woodward, "*Evolutionary project management*". *Computer* **32(10)**, pp. 49–57, Oct 1999.

[74] G. Xie, J. chen, and I. Neamtiu, "*Towards a better understanding of software evolution: An empirical study on open source software*". In *Intl. Conf. Softw. Maintenance*, pp. 51–60, Sep 2009.