Evolution of the Linux Kernel Variability Model

Rafael Lotufo[1], Steven She[1], Thorsten Berger[2],
Krzysztof Czarnecki[1], and Andrzej Wąsowski[3]

[1] University of Waterloo, Ontario
{rlotufo,kczarnec,shshe}@gsd.uwaterloo.ca
[2] University of Leipzig, Germany
berger@informatik.uni-leipzig.de
[3] IT University of Copenhagen, Denmark
wasowski@itu.dk

Abstract. Understanding the challenges faced by real projects in evolving variability models, is a prerequisite for providing adequate support for such undertakings. We study the evolution of a model describing features and configurations in a large product line—the Linux kernel variability model. We analyze this evolution quantitatively and qualitatively.

Our primary finding is that the Linux kernel model appears to evolve surprisingly smoothly. In the analyzed period, the number of features had doubled, and still the structural complexity of the model remained roughly the same. Furthermore, we provide an in-depth look at the effect of the kernel's development methodologies on the evolution of its model. We also include evidence about edit operations applied in practice, evidence of challenges in maintaining large models, and a range of recommendations (and open problems) for builders of modeling tools.

1 Introduction

The cost of variability management in software product lines is meant to be offset by the savings in deployment of product variants over time. Product families with long lifetime and large number of variants should provide a bigger return over time. For these reasons a product line architecture is typically implemented in large projects with a long time horizon. The time horizon and the sheer size of these projects place coping with *scale* and *evolution* as the forefront challenges in successfully running software product lines.

Variability models evolve and grow together with the evolution and growth of the product line itself. Thus realistic feature models are large and complex [1], reflecting the scale of growth and evolution. Nevertheless, evolution of real variability models has not been studied. Multiple authors have been interested in reasoning about feature model editing [2,3], in semantics of feature model refactorings [4], or in synchronizing artifacts in product lines [5,6], which indeed, as we shall see, is a major challenge in maintaining a variability model. However, none of these works was driven by documented challenges faced by practitioners.

We set out to study how feature models evolve, and the main challenges encountered in the process. Do the cross-tree constraints deteriorate or dominate

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 136–150, 2010.

hierarchy over time? Does the number of cross-tree dependencies become unmanageable? Is the model evolved ahead of the source code, along with the source code, or following the source code? We address these and similar questions, hoping to inspire researchers and industries invested in building tools and analysis techniques for variability modeling.

The subject of our study is the Linux feature model. As argued previously [1], the model extracted from Linux Kconfig is, so far, the largest feature model publicly known and freely available. We study the evolution of this model over the last five years, when Linux and the model were already at a mature stage. The model demonstrates that a lasting evolution of a huge product family is feasible and does not necessarily deteriorate the quality of the feature model. Despite the number of features doubling in the studied period, structural and semantic properties of the model have changed only slightly over time, retaining the desirable aspects, such as balanced composition and limited feature interaction.

The main contributions of this work are the following:

- A study of evolution of a real-world, large and mature variability model;
- Evidence of what operations on feature models are performed in practice;
- Evidence of what refactorings are applied to models in practice;
- Evidence of the difficulty for humans to reason about feature constraints;
- Input for designers of tools and techniques supporting model evolution.

We give background on Linux and its configuration language in Section 2. Section 3 justifies the choice of the experiment subject and period, and sketches the experiment design. Section 4 presents and analyzes the collected data. Remaining sections summarize threats to validity, related work, and our conclusions.

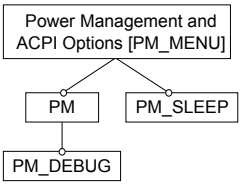
2 The Linux Kernel and Its Variability Model

Born in 1991, the Linux kernel is one of the most mature open source projects as of writing, and continues to be rapidly developed. It remains a crucial component of numerous open and closed source projects, including distributions of the GNU/Linux operating system, mobile phones, netbook computers, network routers, hardware media players and similar appliances. This diversity of applications and users, enforces a highly configurable architecture on the kernel. Indeed Linux kernel is among the largest well documented software product lines studied so far [1,7].

Linux development community comprises both volunteers and paid developers recruiting from more than 200 companies including Red Hat, IBM, Intel, Oracle, Google and Microsoft among others [8]. The maturity of the project manifests in multiple metrics such as the codebase size (exceeding 8 million lines), the number of active developers (600–1200 per release and growing), and the level of activity (up to 10000 patches per release).

Kernel versions numbers are triples: triple 2.6.12 represents a kernel from the 2.6 branch at minor revision number 12. A new minor revision is released every 3 months. All revisions studied in this paper belong to the 2.6 branch, and thus

```
1  menu "Power management and ACPI options"
2    depends on !X86_VOYAGER
3    config PM
4      bool "Power Management support"
5      depends on !IA64_HP_SIM
6    config PM_DEBUG
7      bool "Power Management Debug Support"
8      depends on PM
9    config PM_SLEEP
10     bool
11     depends on SUSPEND || HIBERNATION
12                || XEN_SAVE_RESTORE
13     default y
14 endmenu
```



$PM\_MENU \rightarrow \neg$ X86_VOYAGER
$PM \rightarrow \neg$ IA64_HP_SIM
$PM\_SLEEP \leftrightarrow$
SUSPEND || HIBERNATION ||
XEN_SAVE_RESTORE

Fig. 1. A simple Kconfig model (left) and the corresponding feature model (right)

we will only use minor revision numbers when referring to them (so 12 denotes 2.6.12). We shall use the terms revision, release and version interchangeably.

The Linux kernel contains an explicit feature model (the Linux kernel feature model) expressed in the domain specific language called Kconfig. The Kconfig language was officially merged into revision 2.5.45 in October 2002 [9]. It has been the language for the Linux kernel feature model ever since. Thus, the Linux kernel feature model is a mature model with as much as 8 years of history in its current form (and a good prehistory in predecessor specification languages). We shall analyze the last five years of this history, which span the mature stage of the model evolution, still characterized by an unprecedented growth.

We now present the Kconfig language. Configuration options are known as *configs* in Kconfig. They can be nested under other configs and grouped under *menus*, *menuconfigs* and *choice groups*. The kernel configurator renders the model as a tree of options, which users select to specify the configuration to be built.

Figure 1 shows a fragment of the Linux variability model, containing a menu (line 1) with two Boolean configs as children: PM (lines 3–5) and PM_SLEEP (lines 9–13). Configs are named parameters with a specified type. A boolean config is a choice between presence and absence. All configs in Figure 1 are bool (e.g. line 4). Integer configs specify options such as buffer sizes. String configs specify names of, for example, files or disk partitions. Integer and string configs are *entry-field configs*—shown as editable fields in the configuration tool.

A *depends-on* clause introduces a hard dependency. For example, PM can only be selected if IA64_HP_SIM is not (line 5). Conversely, a *select* clause (not shown) enforces immediate selection of another config when this config is selected by the user. Nesting is inferred by feature ordering and dependency: for example PM_DEBUG is nested under PM (line 8). A *default* clause sets an initial value, which can be overridden by the user. For example, PM_SLEEP defaults to y.

Menus are not optional and are used for grouping, like mandatory non-leaf features in feature models. *Choices* (not shown) group configs, which we call

choice configs, into alternatives—effectively allowing modeling of XOR and OR groups. *Menuconfigs* are menus that can be selected, typically used to enable and disable all descendant configs.

As in [1,9], we interpret the hierarchy of configs, menuconfigs, menus, and choices as the *Linux feature model*. The right part of Figure 1 shows the feature model for the Kconfig example in the left part of the figure. Table 1 maps basic Kconfig concepts to feature modeling concepts. An entry-field config maps to a mandatory feature with an attribute of an appropriate type, integer or string. Conditional menus map to optional features; unconditional menus to mandatory features. We map a choice to a feature with a group containing the choice configs. A mandatory (optional) choice maps to a mandatory (optional) feature with an XOR-group. More details on Kconfig and its interpretation as a feature model are available in [1].

3 The Experiment

3.1 Linux Feature Model as a Subject

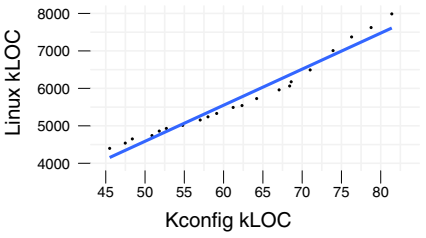
Before we proceed to our experiment, let us address the basic relevance: are the Linux variability model and the selected period of evolution relevant to study?

We analyze the evolution of the Linux kernel feature model between revisions 12 and 32—a period extending over almost 5 years, in which the Linux code base was already large and well established, while still growing rapidly. Meanwhile, the size of the kernel, measured as the number of lines, has doubled. It was also a period of intensive changes to the Kconfig model, since maintenance and evolution of this model follows the source code closely in size and in time.

Figure 2 plots the size of the Linux source code against the size of the Kconfig files (a), and against the number of features declared in Kconfig (b). Since all these measures are growing monotonically with time, the progression of samples from the origin towards the right top corner is ordered by revision numbers. Each point represents one of the 21 revisions between 12 and 32. We observe that in

Table 1. A simplified mapping of Kconfig models to feature models [1]

Kconfig concepts		**Feature modeling concepts**
Boolean config		Optional feature
Entry-field config		Mandatory feature
Conditional menu		Optional feature
Unconditional menu		Mandatory feature
Mandatory Choice		Mandatory feature + (XOR,OR)-group
Optional Choice	➠	Optional feature + (XOR,OR)-group
Config, menu or choice nesting	➠	Sub-feature relation
Visibility conditions, Selects, Constraining defaults	➠	Cross-tree constraint



(a) Total kLOC for Linux against kLOC of Kconfig files

(b) Total kLOC for Linux against the number of features in Kconfig files

Fig. 2. Evolution of number of features and lines of code from revisions 12 to 32

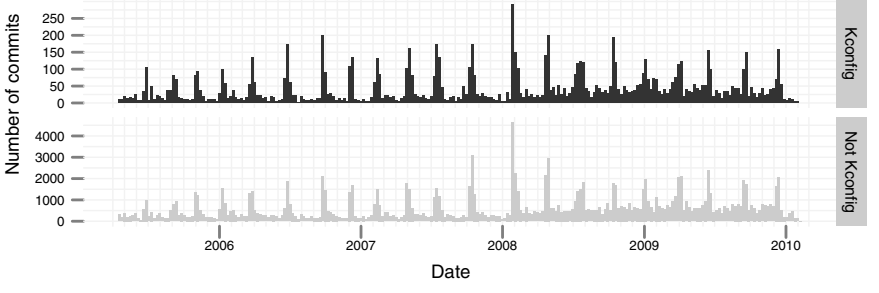


Fig. 3. Number of commits per week that touch Kconfig files compared to number of commits that do not. Each spike matches one of the 21 revisions analyzed.

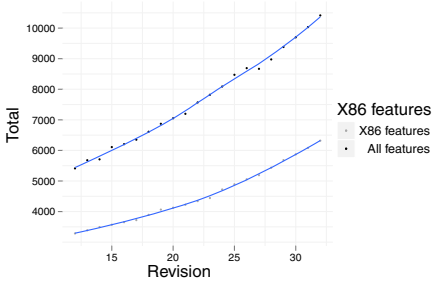
the given period the feature model grows almost linearly with the amount of source code, and its textual representation (Kconfig files).

Figure 3 shows the number of patches added weekly to the Linux source code that modify, and also that do not modify, Kconfig files. Both numbers exhibit almost identical 'heart-beat' patterns, suggesting a causal dependency between changes to the model and to the code.

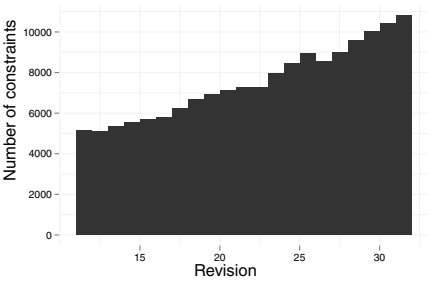
All three diagrams are strong quantitative indications that the development of the Linux kernel is feature-driven, since the source code is modified and grows along with the modifications to and growth of the feature model. This feature-oriented development on large scale makes the Linux model an interesting and relevant subject of investigation. We can expect that challenges faced by Linux maintainers can be exemplary also for other projects of similar maturity.

To scope our investigation, we focus on the model for the x86 architecture, extracted from the main line of development[1]. This scoping does not significantly skew our results, since x86 is the longest supported, the largest and the most widespread architecture of the kernel. We have verified that the x86 feature model exhibits the same pattern of growth as the entire model. For example, see the growth of the number of features in Figure 4a plotted for the entire kernel, and for the x86 architecture. Note that the x86 architecture was created

[1] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git



(a) Total number of features in Linux Kconfig and x86 architecture



(b) # of constraints per revision

Fig. 4. Growth in number of features and number of constraints

in release 24 by merging the 32-bit i386 and the 64-bit x86_64 architectures.[2] From releases 12 to 23 we consider the i386 architecture as the x86 architecture.

3.2 Data Acquisition

Since release 2.6.12 the Linux kernel uses Git (http://git.or.cz) as its version control system. The Git commit history is a series of atomic patches extending over multiple files, each of which contains the commit log, and a detailed explanation of the patch. In the Linux project each patch is reviewed and signed-off by several experts. Since Git allows history rewriting [10], only few patches contain incorrect or misleading information. Thus we consider the Linux Git repository a trustworthy source of information and we limit our attention span to releases in which Git was used. Although historical revisions predating 2.6.12 have been converted to the Git format, one has to keep in mind that they were created using different tools, and thus in different circumstances. To assure quality and consistency of our mining, we chose not to extend our investigation before 2.6.12.

We use a parser [1] extracted from the Linux xconfig configurator to build the feature hierarchy tree. This ensures reliable and consistent interpretation of syntax. We use CLOC (http://cloc.sourceforge.net) to measure code size. Blank lines, comments, and files not recognized as code by CLOC are ignored.

4 Evolution of the Linux Kernel Variability Model

We shall now present and analyze the collected data, dividing it into two parts: Section 4.1 on the macro-scale, and Section 4.2 on the micro-scale.

4.1 Evolution of Model Characteristics

In [1] we have identified and described a number of characteristics of the Linux kernel feature model. We will now analyze how they change over time.

[2] More details at http://kernelnewbies.org/Linux_2_6_24

Model Size. As previously said (see Figure 4a), the number of features of the x86 feature model has almost doubled during the studied period, growing from 3284 in release 12 to 6319 in 32. The growth is steady and uniformly distributed, indicating a regular development pace, and a repetitive development cycle. Also, as seen in Figure 2 this growth is paralleled by the code growth, with a roughly constant feature granularity (measured as average SLOC per feature).

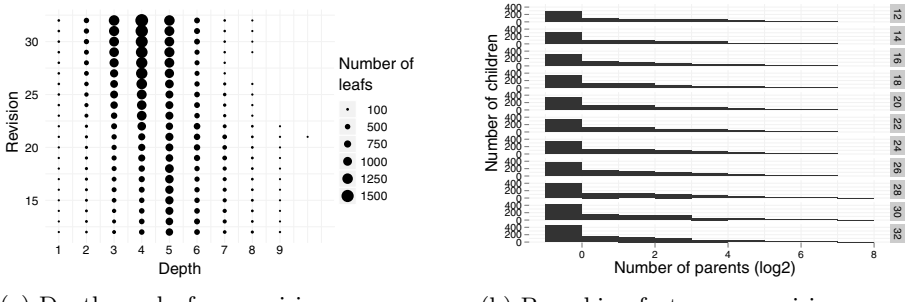
Depth of Leaves. As the hierarchy is the only structuring construct in feature models (and in Kconfig), the growth of the model must necessarily influence either depth or breadth of the hierarchy.

In revision 12 the deepest leaf is at depth 9, and most leaves are at depth 5. Somewhat counter-intuitively, both maximum depth and dominant depth have decreased over time: maximum depth in revision 32 is 8 and most leafs are now at a depth of 4—see Figure 5a. The dominant depth has decreased, even as the number of features at depth 5 continued to increase. So this feature model has been growing in width, not in height.

As reported in [1], the Linux kernel feature model is shallow and has been so at least since revision 12. The Linux project evolves the model in such a way that the basic structure of the hierarchy remains stable over considerable periods of time, despite massive changes to features themselves.

Constraints. In [1] we report that the Linux kernel feature model constraints are mostly of type 'requires'. However we did find considerably many constraints involving more than one feature, with extreme cases of constraints containing up to 22 features. We now examine how these properties change over time.

Figure 4b shows that the number of constraints has increased over time: the amount of constraints in revision 32 is almost double that of revision 12. It is interesting to note that contrary to the belief that the number of constraints grows quadratically with the number of features, the two numbers have grown in the same proportion in the given period. Again, the Linux kernel model demonstrates that it is feasible to construct software architectures and models that only induce constant number of dependencies per feature. In this sense it proves that feature models are a feasible modeling language for large projects.



(a) Depth per leaf per revision

(b) Branching factor, per revision

Fig. 5. Basic characteristics of hierarchy and branching factor across revisions 12–32

Branching Factor. We also measured the branching factor for each of the revisions, in the same manner as in [1] (see Fig. 5b). We found that there was no significant change in the shape of the histogram of children per feature, except that the number of features for each branching factor have increased. For example, there were approximately 300 features with one child, and outliers with 120 children in revision 12. In revision 32 these numbers are 400 and 160.

4.2 Summary of Model Content Changes

We have seen that the feature model has undergone many changes between revision 12 and 32. In particular, the size, average depth and number of constraints were affected. We shall now look deeper into these changes. We will characterize the edits that affected these characteristics, their overall motivation, and the implications for tool developers. For the purpose of this investigation, we define an *edit* to a feature model as a series of changes committed in the same patch.

In order to analyze motivation for individual edits to the model, we have selected a set of 200 uniformly random patches from the Git log, out of 8726, that in the given period touch Kconfig files. We have used this sample for training, to identify six categories of reasons for changes in the Linux model:

New functionality: model modifications when adding new configurable functionality;
Retiring obsolete features: modifications removing functionality from the project.
Clean-up/maintainability: modifications that aim at improving usability and maintenance of the feature model;
Adherence to changes in C code: model modifications reflecting changes made to dependencies in C code in the same patch;
Build fix: reactive modifications that adjust the feature model to reflect changed dependencies in C code in prior patches;
Change variability: adjustments to the set of legal configurations of the feature model without adding code for new functionality.

After defining the above categories, we have independently selected another 200 patches, but this time out of 7384 of those touching Kconfig files used in the x86 architecture model. We classified this sample manually and interpreted the results. We have restricted ourselves to the x86 features, in order to be able to relate the results of this study to characteristics computed for x86 in Section 4.1.

Figure 6 shows the results. In the following paragraphs we discuss each of the categories in detail, outlining its typical edit patterns, the effects it has on the feature model, and the tooling that would be desirable for the given scenario.

New functionality. Close to half of the patches in our sample were related to functionality changes—either adding to, or removing from the kernel. Each of these predominantly simple patches comprises of adding functional C code, updating a Makefile to specify how the new code will be compiled into the kernel, and typically adding one new config with simple constraints and a documenting

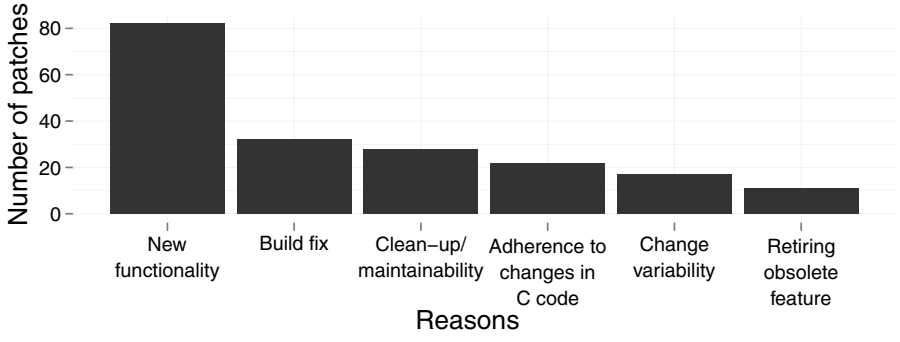


Fig. 6. Reason for edits (sample)

help text. When more than one feature is added, they are typically siblings. Most often these operations do not add further constraints to the model.

Feature additions rarely make intrusive changes to the feature model hierarchy, as almost 87% of all new features are added at leaves. Details are available in Figure 7a, which also shows that more than 50% of new features are added as leaves at levels 3, 4 and 5. Our hypothesis is that this is because the x86 architecture is very mature, and developers add features to existing elements ("slots") of the architecture, without extending the architecture itself.

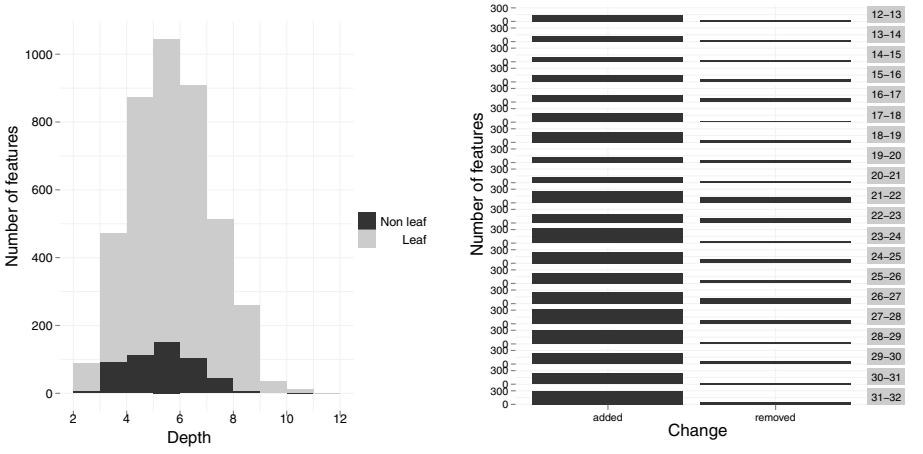
Figure 7b shows the number of features added and removed in consecutive releases. As expected, the number of feature additions in total and per release exceeds that of feature removals. Figure 7b also shows that the number of features added between releases 12 to 23 is much smaller than the additions from release 24 to 32, and correlates well with Figure 3, where we see much higher numbers of commits per week after revision 24 (January 2008).

Thus, most edits to the Linux kernel between revision 12 and 32 add new functionality, new drivers and features, as opposed to performing code and model refactorings. This is consistent with our findings (Figure 6) that almost half of the sampled patches are motivated by inclusion of new functionality and features. This also correlates to the findings of [11] which found that the super-linear growth of the Linux kernel from 1994 to 2001 was due to the growth of driver code, where drivers are typically added as new features.

Retiring obsolete features. As previously shown (Figures 4a, 6 and 7b), removing features is a rare motive for edits. We have found that this mostly happens when features are no longer supported by any developer, or when the feature has been replaced by another, making the former feature obsolete. These operations are the inverse of the operations shown in the previous section, and mostly consist of removing C code, build instructions and the related config from the model.

Notably, the kernel project maintains a formal schedule of retiring features and code, which can be found in the project tree.[3] Every entry in this file describes

[3] The file is `Documentation/feature-removal-schedule.txt`



(a) Depth of added features

(b) # of added, removed and moved features

Fig. 7. Added and removed features for releases 12–32

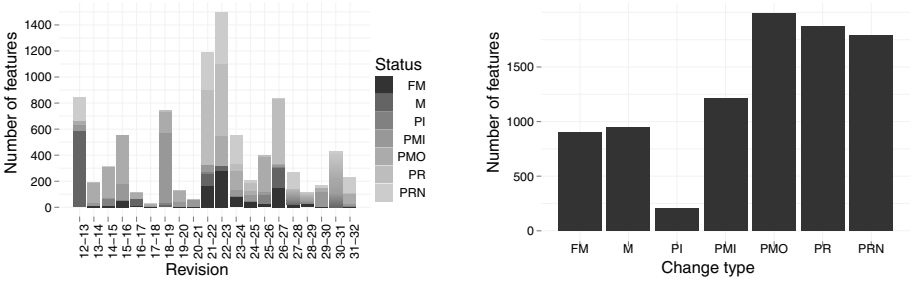
what exactly is removed, why it is happening, and who is performing the removal. This was the first time when the authors of this paper experienced such a formalized and feature-driven (!) process for phasing out code.

Retiring features is not well supported by existing tools and model manipulation techniques. It would be desirable to provide tools that: (a) eliminate features from the model (including from the cross-tree constraints, without affecting the configuration space of the other features, and possibly performing diagnosis about the impact of removal), and (b) use traceability links to find code related to the feature to verify correctness of code retiring.

Clean-up/maintainability. As seen in Fig. 6, developers frequently edit the model to improve its maintenance and usability. These edits typically focus on the end users by improving help text and feature descriptions or by refactoring the hierarchy. Constraint refactorings are a common consequence of hierarchy refactoring in the Kconfig syntax.

We assume that a feature f was subject to hierarchy refactoring if its parent has changed between releases. Let f be a feature that moved from parent p_1 to p_2 between revisions m_1 and m_2. We distinguish seven cases of parent change:

1. *Parent introduction* (PI): p_2 introduced between p_1 and f and p_2 not in m_1;
2. *Parent moved in* (PMI): p_2 introduced between p_1 and f and p_2 exists in m_1;
3. *Parent removal* (PR): p_1 is removed from the feature model (p_1 not in m_2);
4. *Parent move out* (PMO): p_1 is moved (found both in m_1 and m_2);
5. *Parent rename* (PRN): p_1 is renamed: p_1 not in m_2 and p_2 not in m_1;
6. *Feature move* (FM): f is moved from p_1 to p_2 and both exist in m_1 and m_2;
7. *Multiple* (M): a combination of at least two of the above.



(a) Cases of hierarchy per revision (b) Hierarchy refactorings aggregated

Fig. 8. Causes of hierarchy refactoring

Figure 8a shows that there has been significant hierarchy refactoring performed to features during the period, specially in releases 22–23. Curiously, release 22 is considered by the Linux kernel community as a bug-fix release.[4]

Figure 8b reveals that changes of parent are mostly caused by operations on the parent itself, rather than by the explicit moving of a feature. When features are moved, they are moved together with their siblings from a common origin parent to a common destination parent. In fact, for all moves in the period, we found that out of 65 origins, only 4 split its children into more than one destination; and out of 68 destinations, only one came from more than one origin. This suggests that feature model editors, should support moving groups of siblings within a hierarchy (as opposed to only allowing moving subtrees rooted in a single feature to a new place in the hierarchy). Another frequent operation is splicing out features from the hierarchy (to remove them or to move them into another position), without affecting the ancestors and the subtree. To the best of our knowledge neither of operations is directly supported by existing editors.

After further investigation, we found that the underlying reasons for a high level of hierarchy refactoring in releases 22–23 was a consistent replacement of a menu and a config with a menuconfig, consequently eliminating one level of the hierarchy and moving more than 500 features up in the hierarchy. The replacement of menu with menuconfig is made to remove unnecessary mandatory features and replace them with an optional feature capable of enabling/disabling an entire tree hierarchy. This explains the decrease in the average depth over the period, mentioned in Section 4.1.

Adherence to changes in C code. Our sample shows that approximately 15% of edits to the feature model are made together with changes in dependencies in C code. These changes in code are typically code refactoring or bug fixes. The edits to the feature model in 90% of these patches are changes to constraints, following the changes in dependencies in C code.

Build fix. When dependencies in code change and are not immediately reflected in the feature model, developers and users may be unable to successfully compile

[4] http://www.linux-watch.com/news/NS8173766270.html, seen 2010/02-28.

the kernel, and therefore significant development time and user satisfaction is lost. We define a *build fix* to be a delayed adaptation of the model to a change to the source code that appeared in another, earlier patch.

Edits to the feature model in these cases resemble those described in *Adherence to changes in C code*. It is striking that build-fixes are so frequently occurring—clearly indicating need for further research on tools that synchronize the constraints in the model and dependencies in the build system.

Commit logs for changes in constraints indicate that developers do not have enough support for reasoning. Comments range from : "After *carefully examining* the code...", "*As far as I can tell*, selecting ... is redundant" to "we do a select of SPARSEMEM_VMEMMAP ... because ... without SPARSEMEM_VMEMMAP gives us a *hell of broken dependencies that I don't want of fix*" and "it's a *nightmare working out why* CONFIG_PM keeps getting set" (emphasis added). They indicate need for debugging tools for feature models that could demonstrate the impact of edits on the model and on the build system.

Change variability. We have found that there are cases where edits change the configurations with the purpose of adding (or removing) an existing functionality to the feature model, allowing users more configuration options. These operations do not add functional C code; they typically add new configs, make changes to constraints, and add variability to C code by editing `#ifdefs`.

These edits, although few, can be highly complex. Depending on the cross-cutting characteristics of the functionality in question, they may require changes to several different files and locations. For example, commit `9361401` named `'[BLOCK] Make it possible to disable the block layer [try #6]'` required changes to 44 different files and more than 200 constraints.

5 Threats to Validity

External. Our study is based on a single system (Linux). However, we know that this is a mature real world system. As the variability model is an integral part of the Linux kernel, we believe that it should reflect properties of many other long lived models that are successfully evolved, such as operating systems, and control software for embedded systems. We have made initial explorations into the Ecos operating system, which seems to confirm our expectations. Nevertheless, one should not consider our recommendations as representative, since we make them by studying this particular project and not by studying a wide sample of projects.

The Linux development process requires adding features in a way that makes them immediately configurable. As a consequence, it not only enables immediate configurability, but also makes the entire code evolution feature-oriented. Arguably, such a process requires a significant amount of discipline and commitment that may be hard to find in other industrial projects.

Not all projects assume closed and controlled variability model. Many projects are organized in plugin architectures, where variability is managed dynamically using extensions (for example Mozilla Firefox or Eclipse IDE). Our study does not provide any insight into evolution of variability in such projects.

We only look at a fragment of the Linux evolution. We consider this fragment to be relevant since it covers roughly 25% of 20 years long history of Linux. It clearly gives us a glimpse into the evolution of a mature and stable product line.

Internal Validity. Extracting statistical data can introduce errors. We are relying on our own infrastructure for automatic analysis of the Kconfig models. This infrastructure uses the parser extracted from Linux tools, for improved reliability. Also, we are reasonably confident about the quality of the infrastructure, given that we have used it before in another study.

Extracting statistics based on release points may ignore essential information. However we consider any serious fluctuations of our data rather unlikely, should the experiments be carried out at the level of individual patches (partly because our statistics are consistent with each other).

Git allows rewriting histories in order to amend existing commits, for example to add forgotten files and improve comments. Since we study the final version of the history, we might miss some aspects of the evolution that has been rewritten using this capability. However, we believe that this is not a major threat, as the final version is what best reflects the intention of developers. Still, we may be missing some errors and problems appearing in the evolution, if they were corrected using history rewriting. This does not invalidate any of our findings, but may mean that more problems exist in practice.

We use an approximation in interpreting parent change operations above.

Manual classification of edits was feasible and reliable due to excellent comments in Git logs for most of the patches. We increased the robustness of the manual analysis by first running a study on 200 features to identify categories, and then analyzing another set of 200 features selected with uniform probability. An improved study would involve independent cross checking of results.

6 Related Work

The evolution of the Linux kernel between 1994 and 2000 was studied by Godfrey [11], which also found that the Linux architecture is mature and had been growing in a super linear rate, due to growth of driver code. We have found that 3578 patches that modify Kconfig files are driver related. Similarly, Israeli [12] collected several software metrics of the Linux kernel source code from 1994 and 2008, and also observed the functional growth by counting features. Adams studies the evolution of the Linux kernel build system [13] and finds that considerable maintenance to the system is performed to reduce the build complexity, that grows, partly due to the increase in number of features.

Svahnberg and Bosch [14] have studied the evolution of two real software product lines, giving details on the evolution of the architecture and features, closely

related to implementation. They also found that the most common type of changes to the product line is to add, improve or update functionality. Our work, however focuses on the evolution of the model supporting the product line.

Extensive work [2,7,4] addresses issues relevant for detecting edits that break existing configurations and product builds. In [2] an infrastructure is proposed to determine if feature model edits increase, decrease or maintain existing configurations. In [4] a catalog of feature model edits that do not remove existing configurations is presented. Tartler et al. study the Linux kernel [7] and propose an approach to maintain consistency between dependencies in C code and Kconfig.

Work on real case studies on moving to a software product line approach can be found in [15,16,17]. They discuss techniques, processes and tool support needed to make the transition. These works, like [14], focus on product line evolution, not the model, but also suggest that tool support is essential.

7 Conclusion

To the best of our knowledge, this paper is the first to provide empirical evidence of how a large, real world variability model evolves. We have presented the study using the Linux kernel model as our case, collecting quantitative and qualitative data. The following list summarizes the major findings of our work:

- The entire development process is feature driven. In particular the feature model grows together with the code and it is being continuously synchronized with the code. Also the code is systematically retired by eliminating features (and the related implementation). Thus, Linux kernel is a prime example of a mature large scale system managing variability using feature models.
- The model experiences significant growth, in number of features and size. Nevertheless, the dependencies between features only grows linearly with size: the number of features have doubled, but the structural complexity of the model remained roughly the same, indicating a careful software architecture which models features and their dependencies in a sustainable fashion.
- The purpose of most evolution activity is adding new features. The model grows in the process, but only in the width dimension (as opposed to depth). The second largest class of model manipulations are caused by the need to reflect changes in dependencies in source code, in most cases, reactively. Most of the changes at the macro-level are caused by hierarchy refactoring. Constraint refactoring is done for maintenance purposes and is also significant.
- To support evolution, tools should support use cases such as: eliminating features with minimal impact on configuration space, refactoring constraints, propagating dependencies from code to the feature model and tools that allow to manipulate hierarchy easily, while automatically adjusting constraints.

Finally, our investigation proves that maintaining large variability models is feasible and does not necessarily deteriorate the quality of the model. In future work we intend to work on some of the support tools mentioned above.

References

1. She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K.: The variability model of the linux kernel. In: VaMoS, Linz, Austria (2010)
2. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE, pp. 254–264 (2009)
3. Janota, M., Kuzina, V., Wąsowski, A.: Model construction with external constraints: An interactive journey from semantics to syntax. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 431–445. Springer, Heidelberg (2008)
4. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., de Lucena, C.J.P.: Refactoring product lines. In: GPCE, pp. 201–210 (2006)
5. Kästner, C., Apel, S.: Type-checking software product lines - a formal approach, pp. 258–267 (2008)
6. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 31–45. Springer, Heidelberg (2008)
7. Tartler, R., Sincero, J., Schröder-Preikschat, W., Lohmann, D.: Dead or alive: finding zombie features in the linux kernel. In: FOSD, pp. 81–86 (2009)
8. Kroah-Hartman, G., Inc., S.L., Corbet, J., LWN.net, McPherson, A.: Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it (2009)
9. Sincero, J., Schröder-Preikschat, W.: The linux kernel configurator as a feature modeling tool. In: ASPL, pp. 257–260 (2008)
10. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The promises and perils of mining git. In: Mining Software Repositories (2009)
11. Godfrey, M.W., Tu, Q.: Evolution in open source software: A case study. In: ICSM, pp. 131–142 (2000)
12. Israeli, A., Feitelson, D.G.: The Linux kernel as a case study in software evolution. Journal of Systems and Software, 485–501 (2010)
13. Adams, B., De Schutter, K., Tromp, H., De Meuter, W.: The evolution of the Linux build system. ECEASST (2007)
14. Svahnberg, M., Bosch, J.: Evolution in software product lines: two cases. Journal of Software Maintenance: Research and Practice, 391–422 (1999)
15. Dhungana, D., Neumayer, T., Grunbacher, P., Rabiser, R.: Supporting evolution in Model-Based product line engineering. In: SPLC, pp. 319–328 (2008)
16. Hubaux, A., Heymans, P., Benavides, D.: Variability modeling challenges from the trenches of an open source product line re-engineering project. In: SPLC (2008)
17. Jepsen, H.P., Beuche, D.: Running a software product line - standing still is going backwards. In: SPLC (2009)