Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Coevolution of Variability Models and Related Software Artifacts

A Fresh Look at Evolution Patterns in the Linux Kernel

Leonardo Passos · Leopoldo Teixeira · Nicolas Dintzner · Sven Apel · Andrzej Wąsowski · Krzysztof Czarnecki · Paulo Borba · Jianmei Guo

Received: date / Accepted: date

Abstract Variant-rich software systems offer a large degree of customization, allowing users to configure the target system according to their preferences and needs. Facing high degrees of variability, these systems often employ variability models to explicitly capture user-configurable features (e.g., systems options) and the constraints they impose. The explicit representation of features allows them to be referenced in different variation points across different artifacts, enabling the latter to vary according to specific feature selections. In such settings, the evolution of variability models interplays with the evolution of related artifacts, requiring the two to evolve together, or coevolve. Interestingly, little is known about how such coevolution occurs in real-world systems, as existing research has focused mostly on variability evolution as it happens in

L. Passos, K. Czarnecki, and J. Guo
University of Waterloo, Canada
Tel.: +1 519-884-2277
E-mail: {lpassos, gjm, kczarnec}@gsd.uwaterloo.ca

L. Teixeira and P. Borba
Federal University of Pernambuco, Brazil
Tel.: +55 81-2126-8430 ext 4323
E-mail: {lmt,phmb}@cin.ufpe.br

Nicolas Dintzner
Delft University of Technology, The Netherlands
Tel.: +31 6-2315-7647
E-mail: N.J.R.Dintzner@tudelft.nl

S. Apel
University of Passau, Germany
Tel.: +49 851-509-3225
E-mail: apel@uni-passau.de

A. Wąsowski
IT University of Copenhagen, Denmark
Tel.: +45 7218-5086
E-mail: wasowski@itu.dk

variability models only. Furthermore, existing techniques supporting variability evolution are usually validated with randomly-generated variability models or evolution scenarios that do not stem from practice. As the community lacks a deep understanding of how variability evolution occurs in real-world systems and how it relates to the evolution of different kinds of software artifacts, it is not surprising that industry reports existing tools and solutions ineffective, as they do not handle the complexity found in practice. Attempting to mitigate this overall lack of knowledge and to support tool builders with insights on how variability models coevolve with other artifact types, we study a large and complex real-world variant-rich software system: The Linux kernel. Specifically, we extract variability-evolution patterns capturing changes in the variability model of the Linux kernel with subsequent changes in Makefiles and C source code. From the analysis of the patterns, we report on findings concerning evolution principles found in the kernel and we reveal deficiencies in existing tools and theory when handling changes captured by our patterns.

Keywords Variability · Evolution · Software Product Lines · Patterns · Linux

1 Introduction

Variant-rich software systems offer a high degree of configurability, allowing users to tailor the target system according to their preferences and needs. The high degree of configurability arises from the variability of the artifacts of the system, meaning that they can be configured for use in a particular context [25]. Upon configuration, the target system is restructured accordingly, leading to a specific *variant*. Examples of such systems span different domains, including database management systems [5,31,54,55], SOA-based applications [4], operating systems [2,7,9], and industry-based software product lines.[1]

As large and complex variant-rich systems have considerable numbers of points of variabilities, these systems often describe them in terms of *features*, and they employ *variability models* to explicitly capture user-relevant features and the constraints they impose. Features, in this case, denote either functionality chunks (coarse-grained variability) or fine-grained configuration parameters. Features declared in the variability model may then be referenced in related software artifacts (e.g., Makefiles and C source code) by means of explicit *variation points* (e.g., conditional build rules and *ifdef* annotations). This referencing, in turn, allows different artifacts to vary according to specific configurations (feature selections).

Facing a high degree of variability, variation points spread across many software artifacts, making variability pervasive across the system. For illustration, consider the Linux kernel, a successful, large, and complex variant-rich software system. Along with the complexity of its variability model [40,58], which contains over 13,000 features in its latest release (3.9), the kernel has over 95,000 variation points distributed across its source code (comprising over

[1] `http://splc.net/fame.html`

30,000 implementation and header C files) and build files (comprising over 1,800 Makefiles). This pervasiveness of variability is also found in many other variant-rich software systems, including both open-source software [9, 43, 44] and industrial product lines [6].

When variability is spread across different artifacts, variability evolution requires variability models and related software artifacts to evolve together, or to *coevolve*. The interrelation of multiple sources of variability makes variability evolution intricate. A thorough analysis of the evolution of Linux kernel between releases 2.6.32 and 2.6.33, for instance, shows that 35% of the features removed from the variability model continue to exist elsewhere, being merged with other features, renamed, or becoming an integral part of the code base [51].

Interestingly, existing research has focused much of its efforts on variability evolution as it occurs in the variability model, but it has ignored the coevolution of other related artifacts [1, 17, 24, 28, 40, 53, 58, 64]. Moreover, previous research often relies on randomly-generated variability models or evolution scenarios that do not come from real-world systems [24, 28, 64]. The few existing studies covering variability evolution across different artifacts and from real-world systems are based on small case studies, which are unlikely to reflect the complexity typically found in large systems. For instance, Neves et al. [48] study the coevolution of variability models and related artifacts in real software product lines, but their subjects have 40 features, at most. In addition, their analysis is limited to refinement changes (i.e., changes that do not affect the behaviour of the system). This assumption, however, is too restrictive in practice, as feature modification and retirement often occur [51, 52].

The lack of a thorough understanding of how variability evolves in large and complex real-world systems is directly reflected in the quality of existing tools and methodologies. As Babar et al. point out [3], the few existing approaches claiming to support variability evolution are ineffective in practice, as they fail to support variability evolution across different artifacts:

> *"Variability evolves as a result of adding, deleting, or updating variation points and variants. However, we found little support for systematically and sufficiently supporting evolution in variability models and other related artifacts."*
>
> —Babar et al., IEEE Software, 2010.

To better understand how variability models and related software artifacts coevolve, we study the evolution history of a large and complex variant-rich real-world system: The Linux kernel.

Linux is widely used in industry, with an increasing number of companies supporting its development [13]. Due to its complexity, publicly available source code, and practical appeal, researchers often study the Linux kernel to better understand practical issues arising from the maintenance of variant-rich software systems, subsequently deriving tool support that industry can directly benefit from [16, 33, 44, 45, 59, 63]. In our case, we are particularly interested in understanding how developers coevolve the kernel variability model, build

files, and C source code. As these three artifact types also define the structure of other open-source variant-rich systems [9,44] and some industrial product lines [6], we are confident that investigating coevolution of different artifacts in the context of a large, complex, mature, and long-lived software system such as the Linux kernel will provide insights to foster further research that will eventually lead to better tool support and evolution principles.

In previous work [52], we investigate a sample of the Linux kernel evolution history, deriving a catalog of 13 coevolution patterns relative to feature additions and removals. In that catalog, we report a pattern if we find it to be recurrent: the number of change instances matching a pattern must be equal to, at least, 3% of the size of the sample under analysis.

In this paper, we redefine the recurrence notion of our previous work [52] to better align it with state-of-the-art approaches in pattern analysis [22,36,47]. Specifically, recurrence is now measured by two main criteria. First, a pattern must enclose, at least, three instances. Thus, recurrence is given in absolute terms, rather than a percentage of a certain sample size. This equates recurrence exactly as prescribed by the *Rule of Three*, stating that a pattern should only be claimed as such if it has, at least, three distinct instances.[2] The Rule of Three is a common recurrence measure in pattern analysis, and it has been used for the identification of design patterns [36], refactoring opportunities [22], and antipatterns [47]. In addition, an absolute threshold, as stated by the rule, facilitates the identification of patterns across different systems, as recurrence becomes independent of any selected sample size. Second, a pattern must come from three different sources [36]. Adapted to our context, this requires that a pattern is applied by, at least, three distinct contributors (developers), avoiding bias towards any personal style on how to accomplish variability evolution.

With this refined recurrence notion, we reanalyze our original dataset and verify the catalog of our previous work [52]. Furthermore, our new analysis increases the sample by 30%, comprising 268 feature additions and 132 feature removals in total. Given the sampled additions and removals, we analyze their corresponding commits, along with over 250 extra ones to aid our understanding; in total, we analyze 657 commits. The analyzed commits cover changes in the 2.6.26–3.3 release range of the Linux kernel, spanning almost four years of kernel development. The new catalog we offer in this paper contains seven new variability-evolution patterns, four inferred ones (situations that follow from our set of patterns, but that do not have at least three distinct contributors or at least three instances), and a generalization of a previously reported pattern. We also remove one pattern from our earlier catalog, as it has not been applied by at least three different developers, nor could it be inferred.

We claim the following contributions:

– We provide a detailed study of how variability models coevolve with different artifacts in the context of a large and complex variant-rich software system: The Linux kernel.

[2] See also: `http://c2.com/cgi/wiki?RuleOfThree`

- We define a taxonomy for the coevolution of variability models, build files, and C source code, resulting from the addition or removal of features in the variability model. We organize the proposed taxonomy as a catalog of variability-evolution patterns.
- We devise a repeatable methodology that allows others to recover patterns in systems other than the Linux kernel. For instance, future work may apply our methodology in systems that have a similar structure as found in the Linux kernel, including open-source variant-rich systems [9,44] and industrial product lines [6].
- We identify a set of principles guiding how Linux kernel developers employ *ifdef* annotations when encoding the kernel's compile-time variability. As these principles ease kernel maintenance and evolution, they are also beneficial for other variant-rich software systems that also rely on *ifdef* annotations.
- We present empirical evidence that some evolution scenarios captured by our patterns cannot be correctly handled by state-of-the-art variability evolution techniques.
- We provide empirical evidence for the need of a new theory for software-product-line evolution. While many of our patterns are captured by the existing theory of software-product-line refinement [10], feature-retirement patterns are not. Since the latter are too frequent to be ignored, a new theory should be devised to account feature retirement.
- Based on our catalog of patterns, we formulate a research agenda outlining future research.

2 Background

This section explains how variability spreads across different artifacts of the Linux kernel. We also introduce a notation for describing patterns.

2.1 The Three Spaces of the Linux Kernel

Variability in the Linux kernel is present in three spaces: *variability model*, *mapping*, and *implementation*. Such structure is not exclusive to the kernel, as it is also found in other open-source variant-rich systems [9,44] and industrial product lines [6]. Following the steps in Figure 1, we describe how each of these spaces works and how they are connected.

Variability Model. The Linux kernel variability model comprises a set of files written in the Kconfig language.[3] A configurator renders (step 1) a tree of features from Kconfig files that are available for the user's platform (i.e., processor family). From it, users select features that should be present in the resulting kernel (step 2).

[3] `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`

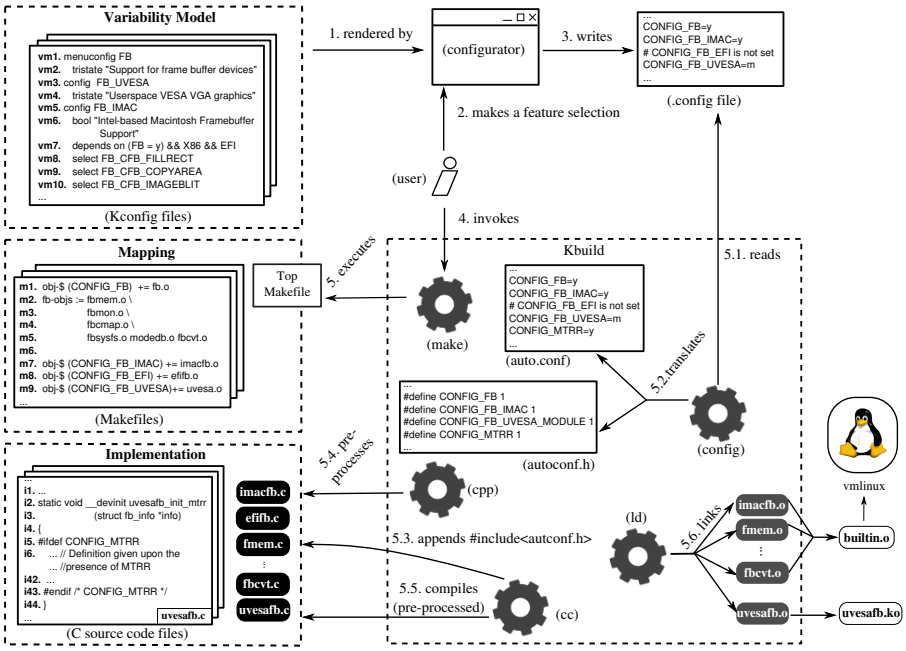


Fig. 1: The three spaces in the Linux kernel and their interaction with Kbuild

As shown in the excerpt of the variability model in Figure 1, features in Kconfig are represented mostly by config declarations (lines vm3 and vm5). In our example, FB (the parent of all frame-buffer-related features)[4] and FB_UVESA (a generic frame-buffer driver) are tristate features (lines vm2 and vm4). They can be absent (n) or present either as dynamically loadable kernel modules (m) or by being statically compiled into the resulting kernel (y). Boolean features are also possible (line vm6), assuming either y or n as value. Other types include integer and strings (not shown).

In Kconfig, features may contain attributes. The prompt attribute is a short text describing the feature (lines vm2, vm4 and vm6). The configurator uses the prompt to render feature nodes in the hierarchy (the absence of a prompt makes a feature invisible to users). A default attribute (not shown) provides an initial value of the corresponding feature, which can be later changed during configuration. Two specific attributes define cross-tree constraints: depends on and selects. The depends on attribute (line vm7) allows writing a dependency stated as a condition that must be satisfied to allow users to select the feature with this attribute. A select attribute is a reverse dependency that enforces the immediate selection of one or more target features. For example, selecting FB_IMAC causes the immediate selection of FB_CFB_FILLRECT, FB_CFB_COPYAREA, and FB_CFB_IMAGEBLIT (lines vm8–vm10).

[4] https://www.kernel.org/doc/Documentation/fb/framebuffer.txt

Once the user finishes the selection, the configuration is saved. The configurator then writes a .config file (step 3), containing a sequence of feature-name=value lines. In this file, feature names are prefixed with CONFIG_.

Mapping. In the Linux kernel, the mapping between features and compilation units occurs mostly inside Makefiles. Kbuild, the kernel build infrastructure,[5] controls the whole compilation process of the kernel. To build a kernel image according to a given configuration, users invoke make (step 4), which triggers the execution of the top Makefile at the root of the Linux kernel source code tree (step 5). The top Makefile then invokes config, which in turn reads the configuration file (step 5.1) and translates it to two other files (step 5.2): auto.conf, later used by make, and autoconf.h, later used by the C pre-processor (cpp).

The top Makefile controls vmlinux (the resident kernel image) and the kernel loadable modules. To build vmlinux, Kbuild first builds all the object files stored in core-y, libs-y, drivers-y, and net-y variables, as stated in the top Makefile:

```
1 vmlinux := $(core-y) $(libs-y) $(drivers-y) $(net-y) ...
2 ...
3 drivers-y += drivers/ main/
```

These variables denote lists of object files to which further elements can be appended. When appending directories (line 3 above), Kbuild recursively runs the Makefile in each of the listed directories and generates all objects of a special list: obj-y (similarly, a list obj-m is kept for dynamically loadable modules). Objects are conditionally added to such a list by replacing y with a feature name. As shown in the Makefile of Figure 1 (line m7), imacfb.o is added to obj-y if FB_IMAC is set to be y in the auto.conf file (the same applies to FB_EFI and FB_UVESA, lines m8–m9). Kbuild attempts to compile object files by locating a corresponding C file with a matching name. If such file does not exist, Kbuild uses a list named after the object file and suffixed with either -y or -objs. In our example, the FB feature is associated with the set of objects in the fb-objs list (lines m2–m5 in Figure 1); there is no fb.c file in the Makefile's directory.

Implementation. Variability in the source code is expressed in terms of conditional compilation macro directives (*ifdefs*). In the C pre-processor, an *ifdef* is either an `#ifdef`, `#ifndef`, `#if`, or `#elif`. The conditions of each of these macro directives are essentially Boolean expressions over feature names, guarding whether certain source code fragments should be compiled.

Prior to compilation, Kbuild adds an inclusion directive to autoconf.h in each target source file (step 5.3). This header file contains macro definitions for the features selected during configuration. It is encoded as follows: all features in the .config file result in pre-processor symbols with the same name; tristate features selected as modules are suffixed with _MODULE; macros of selected

[5] https://www.kernel.org/doc/Documentation/kbuild/

Boolean/tristate features are set to 1; integer/string features, if present, lead to macros whose values match those given during configuration.

Given the macro definitions in autoconf.h, the C pre-processor evaluates all code guards directives, deciding which code blocks to include and which to remove (step 5.4). Then, the C compiler compiles the resulting code (step 5.5). From the example configuration in Figure 1, pre-processing uvesafb.c results in a non-empty body of the __devinit uvesafb_init_mtrr function (lines i6–i42), as CONFIG_MTTR is a defined macro in autoconf.h.

The last step in the compilation process links the object files in obj-y, merging them into a built-in.o file (step 5.6). This file is later linked into vmlinux by the parent Makefile. Similarly, tristate features set to m, after linkage, result in loadable kernel objects (.ko file).

2.2 Patterns and Notation

An evolution pattern summarizes changes in each space and shows how the spaces coevolve. Consider a particular instance of a merge pattern that operates on two framebuffer-related features, which were presented in the previous section: FB_IMAC and FB_EFI. Both features are children of FB. Due to their similarity, developers decide to merge the two features, adding the capabilities of FB_IMAC into the implementation of FB_EFI. To avoid capability redundancy, developers remove FB_IMAC from the variability model, mapping and implementation.[6]

The described change is captured by the pattern in Figure 2. A pattern denotes a transition from a *before-state* to a state after the application of the prescribed change—*after-state*. The transition is represented by an arrow (shown in the middle); the before-state is on the left of the arrow; the after-state follows it. In each state, the pattern captures key characteristics in the variability model, build files, and source code.

We express the variability model in a FODA-based notation, together with the set of the existing cross-tree constraints (i.e., CTC). Since FODA [29] is a simple, intuitive and widespread notation praised by both researchers and practitioners [8], we can abstract over many specific details of Kconfig, while reaching a larger audience. In the before-state of Figure 2, two optional sibling features exist: f_1 (matches FB_IMAC) and f_2 (matches FB_EFI). To explicitly report that these features are visible (promptable) during configuration, we use a corresponding attribute (shown inside square brackets).

We capture the mapping M as a sequence of build rules defined by the following syntax:

$$M ::= \langle R^+ \rangle$$
$$R ::= (E, R, R) \mid \text{compilation unit}^+ \mid \text{directory}^+ \mid \text{compilation flag}^+ \mid \epsilon$$

In a conditional build rule (e, r_1, r_2), e is an expression E over feature names;

[6] See http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=7c08c9ae

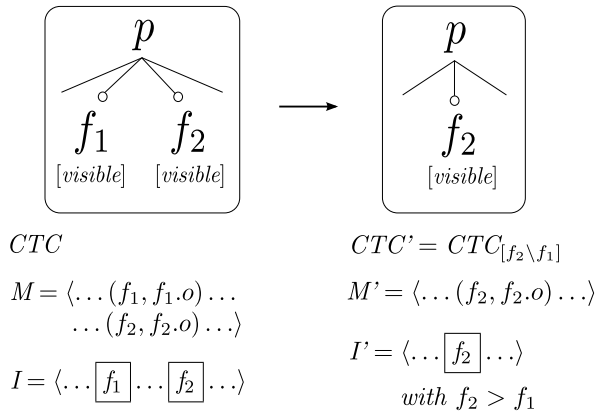


Fig. 2: Definition of Merge Visible Optional Feature into Sibling (MVOFS)

r_1 is another build rule R executed in case e evaluates to true; and r_2 is an alternative build rule for the case e does not hold. The shorthand form (e, r_1) is used when r_2 is empty. Unconditional rules are either a sequence of compilation units, a non-empty list of directories, one or more compilation flags, or an empty rule. The pattern in Figure 2 shows two build rules: $(f_1, f_1.o)$ and $(f_2, f_2.o)$, stating that the presence of f_1 and f_2 triggers the compilation and linkage of their corresponding compilation units (imacfb.o and efifb.o in the example). For simplicity, this representation does not distinguish dynamically loadable modules from objects to be statically linked against the kernel.

Similarly to the mapping space, we capture the implementation (I) as a sequence of code block triples (e, c_1, c_2), where e is a macro-based expression over feature names and c_1 and c_2 are themselves code block triples. As before, simplifications are possible: c denotes an unconditional code block and (e, c_1) is a conditionally compiled code block without an alternative. In case an entire compilation unit implements a feature, we draw a square in the code space (e.g., matching imacfb.c and efifb.c, respectively).

In all spaces, we use ellipses ("...") to ignore unrelated elements that do not affect the features under analysis.

In the after-state of the merge pattern in Figure 2, f_1 is removed from all three spaces (removal is generally denoted by omitting elements previously shown in the before-state). The set of cross-tree constraints is then rewritten (CTC') such that every reference to f_1 becomes a reference to f_2. Besides referential integrity, such rewrite guarantees that all constraints imposed by f_1 are now imposed by f_2 as well (no constraint is lost). Furthermore, the compilation unit of f_2 continues to support the capabilities of f_1, plus its own, which we denote as $f_2 > f_1$.

3 Methodology

We build a catalog of patterns by analyzing commit patches (textual diffs) that change the variability model by either adding or removing feature names. We then keep track of how the mapping and implementation spaces change as a result.

To scope our analysis, we focus on the x86 architecture of the Linux kernel, as the variability model of the x86 architecture follows the same growth pattern of the variability model of the whole kernel [40]. Next, we describe the methodology for data collection, followed by how we identify patterns. All the collected data, its analyses, and the custom underlying infrastructure are available at a supplementary site.[7]

3.1 Data Collection

We collect the entire set of added and removed features by calculating the feature set difference of the variability models of consecutive stable kernel releases. The union of all added features comprise the *additions population*; likewise, the *removals population* is given by the union of all removed feature names. To list the features in the variability model of a given release, we extract the Kconfig infrastructure shipped in the Linux kernel source code. Currently, our infrastructure can process Kconfig files in any version starting from the kernel release 2.6.26, up to 3.3, the latest release available when we first collected patterns.

The size of the additions population in the given release range (4,112) is four times bigger than the size of the removals sample (1,002). These numbers are consistent with other works [17, 40], which show that feature additions in the Linux kernel exceed feature removals.

From the population data, we select two random samples: one comprising 6.5% (268) of all feature additions, and another with 13% (132) of all feature removals. These samples extend the original ones used when extracting the first version of our catalog [52], adding 30% new commits relative to feature additions (62) and removals (31).

An entry in the additions sample is a pair of the form (f, r_{i+1}), where r_{i+1} adds a feature named f that does not exist in the previous stable release r_i. An entry (f, r_{i+1}) in the removals sample mean that release r_{i+1} no longer contains f, although r_i does. A feature f in either of the entries is referred as *primary feature*—a primary object in our investigation.

To obtain the patch adding or removing a primary feature, we must first locate its corresponding commit, referred to as *primary commit*. To that end, we use a custom-made tool [49] to create a relational database from the Linux kernel Git commit history.[8] Figure 3 depicts how the database is populated. First (step 1), we enumerate all stable releases saved in the commit history,

[7] `http://gsd.uwaterloo.ca/coevolution-patterns`

[8] See `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`

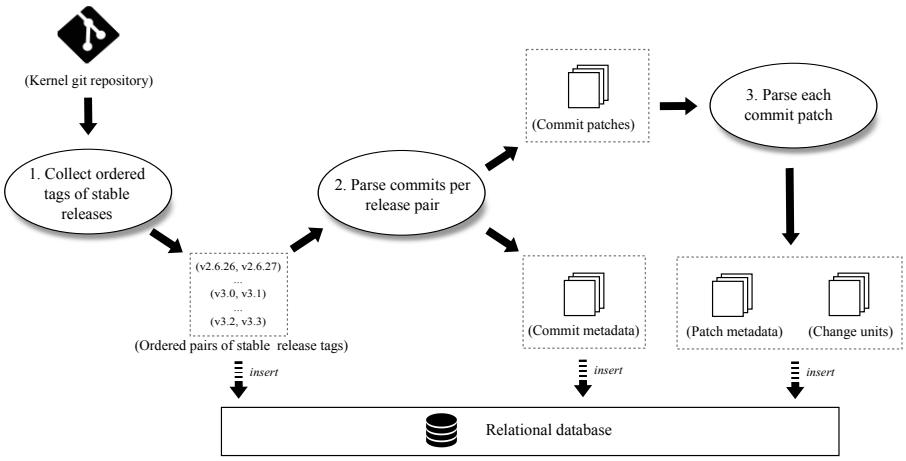


Fig. 3: Database creation process

storing them as ordered sequential release pairs of the form (r_i, r_{i+1}). In step 2, we parse all commits between the releases of each release pair, storing the commit author name and email, the commit message, the commit hash, etc. Next (step 3), we parse the patch of each commit from step 2, saving associated metadata (e.g., the name of the changed file, whether the file is new, removed, or renamed, etc.) and any *feature change units*. A feature change unit is a change that adds or removes a feature name in a Kconfig file. For each change unit in our database, we also store the name of the feature it adds or removes. In all steps, we link data accordingly: each patch metadata and change unit record links to a corresponding commit record, which in turn, links to a specific release pair.

With the database in place, retrieving the primary commit of a primary feature becomes a simple matter of issuing an SQL-query: if a feature f is in the feature set difference of r_{i+1} and r_i, then there exists a primary commit with a change unit adding f. Such commit, in turn, associates with the release pair (r_i, r_{i+1}). Likewise, if f is in the difference of the feature sets of r_i and r_{i+1}, then there exists a primary commit with a change unit removing f. As before, the retrieved commit associates with the release pair (r_i, r_{i+1}).

In the database, a primary commit associates with one or more primary features. Primary features may also have two or more associated primary commits, but we restrict it to be exactly one to facilitate analysis. Taking f as primary feature, we find the following cases that lead to two or more primary commits in the target population:

T1 In addition to x86, f is also in the namespace of other architectures (e.g., sparc, powerpc, etc.), being declared in Kconfig files specific to such CPUs. Therefore, adding or removing f happens in all architectures that support it, having different commits for different architectures (generally,

one per architecture type). When facing multiple commits targeting different architectures, we select the one concerning x86 (our scope of analysis).

T2 A commit adds f, another removes it (e.g., by reverting the first change), and a third adds f again. Likewise, a commit may remove f, a second add it, and a third remove it again. In both cases, we take the primary commit to be the last one in the series, regardless of which sample f originates from.

T3 A commit adds f to its own Kconfig file, which is then included by a parent Kconfig file. Later, another commit replaces the inclusion instruction by the declaration of f itself (another addition). In such situation, we take the first commit, as the second does not affect the namespace; rather, it only replaces f's declaration.

T4 A commit first adds f, followed by another commit creating an additional configuration option f (in Kconfig, it is possible for a feature to be declared twice). Similar to the previous case, the namespace is not changed. As before, we take the first commit as the primary one.

T5 Due to the distributive nature of the kernel development, patches may be submitted more than once. Consequently, different commits may have equal patches. For example, a patch submitted to the kernel mailing list may be accepted by a developer, who commits it to his local copy of the kernel repository. Prior to pushing it to the remote site, the developer pulls from the remote copy to retrieve any updates. Meanwhile, another developer also accepts the change, and prior to pushing it, he also performs a pull to fetch any remote updates. Note that both pulls do not retrieve the accepted change, as it has not been pushed by either developer. Then, the second developer pushes his changes, followed by the push of the first developer. As a result, the remote repository now has two exact patches, each with a different commit hash.

T6 There are two or more commits removing f, with each commit holding a different patch. As an example, consider the case where a commit copies f to a new location in the repository, resulting in a duplicate declaration. A new feature is then introduced, generalizing the capabilities of f. As the generalized feature supersedes the original and the copied features, both must be removed. The developer, however, separates such removal in two commits. The first one contains the removal of the original feature; the second commit contains the patch adding the generalized feature, together with the removal of f's copy. When facing multiple removals, we take the latest one. Likewise, it also happens that two different commits add a feature f in distinct ways. For example, a developer sends to the mailing list a patch adding f, which eventually gets accepted. Later to his first submission, the same developer re-submits the patch with further enhancements.

In the kernel repository, feature additions and removals that link to multiple primary commits are infrequent. In our samples, we only find three additions

Release range: 2.6.26 to 3.3	
Nbr. of commits	176,449
↪ Nbr. of commits changing Kconfig files	10,205
↪ Nbr. of commits adding/removing features	5,704
↪ Nbr. of distinct primary commits in our two samples	359

Table 1: Commit statistics

(two cases of T1 and one case of T5) and two commits removing the same primary feature (T5).

Forcing a primary feature to have exactly one primary commit means that we will work with the same number of primary commits as our sample sizes; hence, we collect 268 and 132 primary commits relative to added and removed features, respectively. Since some primary commits concern more than one primary feature, the number of distinct primary commits (359) is lower than the sum of the two sample sizes. Table 1 puts these statistics into context.[9] The number of distinct commits in our two samples equals to 6% of all commits that either add or remove features. The latter, in turn, is a subset of the commits that necessarily change Kconfig files, representing 56% of all the commits in that set. Commits that necessarily change Kconfig files are a particular piece of the kernel evolution history, accounting for approximately 6% of all commits in the given release range. Overall, the two samples cover 0.2% of all commits between releases 2.6.26 and 3.3.

Once all primary commits are known, we proceed to extract evolution patterns.

3.2 Pattern Extraction

We apply a multiple step analysis to extract the evolution pattern of a primary feature. As the primary commit only guarantees to retrieve changes in the variability model (changes in other spaces may be in other commits), we rely on a *commit window* to expand the search scope for changes in related artifacts.

A commit window is a sequence of commits that in addition to the primary commit, may include commits preceding or following the primary one. To exemplify a commit window, consider the addition of the CAPTURE_DAVINCI_DM64X feature.[10] As shown in Figure 4, the primary commit (highlighted in gray) is part of a sequence of commits changing the V4L and DVB subsystems,[11] as stated in the commit log messages. The primary commit patch is shown in Figure 5. A patch is a textual diff recording added (prefixed with "+") and removed lines (prefixed with "-"). Lines without prefix provide context to ease understanding. In the example, the primary commit

[9] The numbers in the table do not account for commits that merge branches.

[10] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=89803d83`

[11] V4L/DVB: Video for Linux/Digital Video Broadcasting

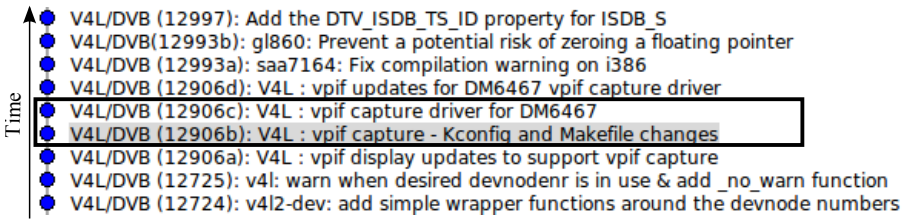


Fig. 4: Commit window example

adds a Kconfig entry (Figure 5, lines 8–11) and a new build rule to compile vpif_capture.c (line 15). Such compilation unit, however, is not added in the primary commit. In that case, we set to expand the commit window to the point where such an addition occurs (if it occurs). The commit following the primary one adds vpif_capture.c; thus, we expand the commit window to include the commit above the primary one. The resulting commit window is shown as a black rectangle in Figure 4.

Strictly, the boundaries of a commit window are only limited by the total number of commits in the evolution history. Furthermore, selecting which commits should be part of a commit window is ultimately a subjective process.

To mitigate subjectivity, we expand a commit window by including commits that have the same commit message *label* as the primary one, and that necessarily preceded or follow it. For example, in Figure 4, all commits changing the V4L and DVB subsystems are labelled with "VL4/DVB", and thus, are potential candidates to be included in the resulting commit window. Following sequences of commits sharing the same label, however, does not necessarily retrieve commits related to the primary feature under investigation (e.g., it may include commits relative to a sibling feature of the primary feature, both belonging to the same part of the kernel). To avoid large windows with unrelated commits, we define four main expansion rules for including commits sharing the same label of a primary commit:

E1 Include commits that add/remove compilation units known to be mapped to the primary feature.
E2 Include commits whose changes affect files mapped to the primary feature.
E3 Include commits whose changes add/remove compile-time variation points that reference the primary feature.
E4 Include commits that modify the declaration of the primary feature in the variability model.

Initially, we apply these rules to expand the commit windows of features in the additions sample only. Starting with the primary commit, we allow a commit window to grow as large as needed, but stop its expansion whenever we meet one of the following boundary conditions: (a) the commits in the current window provide enough context to understand the changes related to the primary feature. For instance, to understand the addition of CAPTURE_DAVINCI_DM64X we are only required to extend the commit window up to the point where

```
drivers/media/video/Kconfig

config DISPLAY_DAVINCI_DM646X_EVM
   help
-     Support for DaVinci based display device.
+     Support for DM6467 based display device.

+config CAPTURE_DAVINCI_DM646X_EVM
+   tristate "DM646x EVM Video Capture"
+   depends on VIDEO_DEV && MACH_DAVINCI_DM6467_EVM
+   ...

drivers/media/video/davinci/Makefile

+obj-$(CONFIG_CAPTURE_DAVINCI_DM646X_EVM) += vpif_capture.o
```

Fig. 5: Patch adding the Davinci D6467 driver (primary commit)

vpif_capture.c is added, but not further; (b) we reach a large sequence of commits that do not share the same label as the primary commit. In this case, we consider the change of the primary feature to be over. The rationale of first expanding commit windows of features in the addition sample follows from our assumption that commit windows of features in the removals sample are likely to be smaller; if true, the maximum commit window size in the additions sample works as an upper bound for the commit window size of features in the removals sample. Our assumption relies on the fact that removing features should be done at once, in a single commit, as developers should not leave dead code behind, nor break the system compilation. Additions, on the other hand, may span more than a single commit, as adding incremental chunks agrees with Git's principle *commit early, commit often*.[12]

We find that commit windows of added primary features have at most 28 commits, although in most cases it has a single one (the primary commit). For defining the commit windows of removed features, we conservatively increase the 28-limit to 40, as an attempt to avoid loosing any commits. Upon the validity of our previously stated assumption, however, commit windows in the removals sample should never reach such a limit. In fact, they do not. After applying the four expansion rules, while respecting boundary conditions and a maximum commit window size of 40, we find that almost every commit window in the removals sample has size one. Few commit windows (6) have more than one commit; three commit windows have two commits, while the remaining three have four, five, and 14 commits, respectively. Overall, commit windows are small in both samples (see Figure 6). In the additions sample, an average commit window has 1.9 commits, whereas in the removals sample, the average

[12] http://sethrobertson.github.io/GitBestPractices/

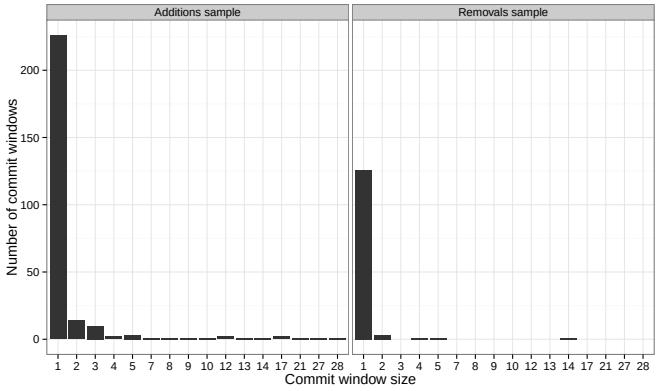


Fig. 6: Commit window sizes

is 1.2.[13] In both samples, the median commit window size is one. Therefore, in the case of the Linux kernel, determining the size of commit windows is not difficult, as a typical commit window contains only the primary commit of the feature under investigation.

Within each retrieved commit window, we move to inspect all the changes it contains, initially classifying it as *addition*, *removal*, *split*, *merge* or *rename* of the primary feature. Windows with the same category are then clustered together. Note that classifying commit windows require us to ignore changes unrelated to the primary feature. Lines 5–6 in Figure 5 show a simple example. More complex unrelated changes occur when a commit window contains patches that, in addition to the primary feature, also add or remove other features. In this case, we set focus on patch parts that explicitly associate with the primary feature (e.g., a code fragment guarded by an *ifdef* condition referring to primary feature, a C file whose compilation depends on selecting the primary feature, etc.), or that relate to it as a consequence of the change under investigation (e.g., a new *ifdef* condition is created for a new feature, which in turn, results from the rename of the primary one).

The relevant changes inside each window are then taken as a whole, which we capture as a before-state and after-state. At this stage, we create specialized subcategories to represent the changes and their similarity in terms of how they affect specific characteristics of primary features and their cross-tree constraints. Such characteristics include, but are not limited to:

a) Visibility: Feature is promptable in the configurator or not;
b) Type: Whether the feature is a *switch* (i.e., Boolean/tristate) or a *value-based* feature (int/string) [9];

[13] These values are calculated as follows: for the additions sample, we take the accumulated count of all its commit windows (502), and divide it by the number of added primary features (268). Likewise, in the removals sample, we sum the count of commits across all commit windows (155), and divide it by the number of removed primary features (132).

c) Computed defaults;
d) Mandatory;
e) Whether the feature causes the addition of compile-time variation points, and in which spaces;
f) Whether the feature contains associated compilation units;
g) Whether the feature adds compilation flags.

We then re-cluster results accordingly and discard clusters with less than three instances, or clusters respecting such threshold, but with less than three distinct contributors. Different from our initial analysis [52], these two key criteria conform to state-of-the-art pattern analysis [22,36,47] and they make the recurrence measure of a pattern independent of the sample size. To differentiate among contributors, we use the contributor's name and email, as recorded in the metadata of each commit. Once we cannot further subcategorize clusters, we set to extract a pattern that explains the changes in the commit windows of each obtained cluster.

In total, we examine 657 commits in all commit windows, where 502 relate to features in the additions sample and the remaining 155 to features in the removals sample.[14] In some cases, however, we cannot derive a full understanding of the changes relative to a primary feature. As an example, consider the addition of the NEED_PER_CPU_KM feature to the kernel memory management subsystem.[15] Figures 7 and 8 show the addition's primary commit (highlighted in gray) and its corresponding patch fragment, respectively. Since the newly added feature is a computed (it is assigned its default value upon the validity of its depend on clause) and invisible feature, users cannot configure it directly. Thus, the computed value of NEED_PER_CPU_KM must be referred elsewhere for the feature to be useful. However, expanding the initial commit window to include commits sharing the same label of the primary commit (shown as a dashed rectangle in Figure 7) does not show any reference addition. Hence, as we cannot fully understand the change in place, we exclude NEED_PER_CPU_KM from further analysis. Overall, when facing doubt, we exclude 4.5% (12) of the features in the additions sample; in the removals sample, the exclusion rate is 8.3% (11).

Following the described methodology, four authors participated in the extraction process, namely A_1–A_4. Authors A_1 and A_4 are proficient Linux users with past experience in the analysis of feature evolution in the Linux kernel [17,49,51,52]; author A_2 has expertise in variability model evolution [24], while A_3 has previously investigated evolution patterns in small-sized software product lines [48]. Table 2 summarizes the role of each author.

In the analyses of the original sample of [52], authors A_1 and A_2 were responsible for extracting patterns (indicated with an 'E' in the corresponding table cell). As extracting patterns requires human analysis (e.g., establishing the

[14] These numbers are obtained by summing the size of each commit window in the corresponding samples.
[15] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=bbddff05`

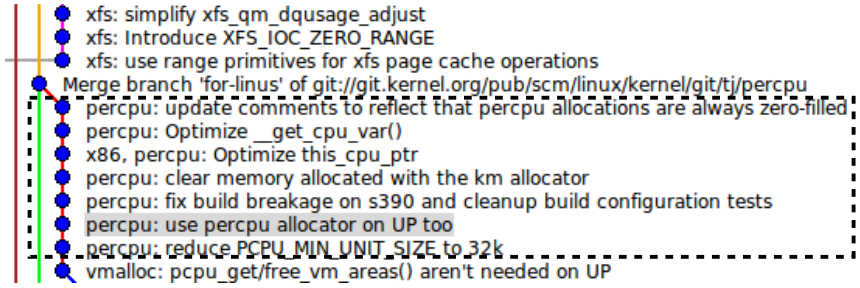


Fig. 7: Commits changing the kernel memory-based chunk allocation

```
1  mm/Kconfig
2
3  +config NEED_PER_CPU_KM
4  +        depends on !SMP
5  +        bool
6  +        default y
7
8  mm/Makefile
9
10 -ifdef CONFIG_SMP
11 -obj-y += percpu.o
12 -else
13 -obj-y += percpu_up.o
14 -endif
15
16 mm/percpu-km.c
17
18 -#ifdef CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK
19 +#if defined(CONFIG_SMP) && \
20   defined(CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK)
21  #error "contiguous percpu allocation is incompatible..."
22  #endif
23
```

Fig. 8: Patch adding NEED_PER_CPU_KM

boundaries of a commit window and defining the subcategories for clustering), A_1 and A_2 worked closely, discussing any arising issue, in addition to reviewing the results of one another (shown with an 'R' in the corresponding table cell). After the extraction of A_1 and A_2, A_3 reviewed their joint work (shown as R(A_1, A_2) in the table), pointing out possible inaccuracies. A fourth review was performed by A_4. Each inaccuracy resulting from the review of A_3 and A_4 was discussed among the authors, which in turn agreed on the final form of the extracted patterns. In the analyses of the 30%-extension of the original

Author	Original sample		Extension sample	
	Additions (206)	**Removals (101)**	**Additions (62)**	**Removals (31)**
A_1	E + R(A_2)	E + R(A_2)	E	E
A_2	E + R(A_1)	E + R(A_1)	–	–
A_3	R(A_1, A_2)	R(A_1, A_2)	R(A_1)	R(A_1)
A_4	R(A_1, A_2, A_3)	R(A_1, A_2, A_3)	R(A_1, A_2, A_3)	R(A_1, A_2, A_3)

Table 2: Activities performed by each author in each sample (E: Extracting patterns; R(A): Review of the patterns extracted/reviewed by author A; R(A, B): Review of the patterns extracted/reviewed by authors A and B)

sample, only A_1 extracted patterns, followed by a review of A_3 and A_4. As before, in the case of inconsistencies, the authors discussed them and reached a final agreement on the correct form of the reported patterns.[16]

3.3 Pattern Inference

After collecting patterns, we compare them to our previous catalog in [52], and label the extracted patterns as follows:

- 'O' (original): The pattern is as reported in the original catalog;
- 'N' (new): The pattern is new, and it is not reported in the previous catalog;
- 'G' (generalization): The extracted pattern results from generalizing a pattern in our previous catalog.

Patterns marked as 'O', 'N', or 'G' have at least three instances in the analyzed samples, with a minimum of three distinct contributors. However, even when either one or both of these two conditions are not satisfied, some patterns can still be inferred from our catalog. We point two specific inference rules:

I1 There exists a pattern adding a given feature, but no inverse pattern exists in the removals sample. From the fact that every added feature should be eventually removed, and that such removal can be achieved by simply following the opposite steps performed when adding the feature, we take the inverse of any addition pattern to be an inferred removal if it is not already in the catalog.

I2 There exists a pattern in the removals sample, but an inverse pattern is not reported in the additions sample. From the rationale that a feature can only be removed if it is first added, and that such addition can be achieved

[16] When reviewing the original and extended samples, A_3 indicated five possible inconsistencies: three were minor comments, whereas for the remaining two, A_3 did not agree that they were instances of a particular pattern. Upon further clarification, A_3 agreed that the two instances were indeed related to the pattern in question. A_4, in turn, pointed out 7% and 14% inconsistencies in the classification of the primary features in the additions and removals samples, respectively. Almost all inconsistencies were confirmed, and the patterns' frequency were changed accordingly.

by following the inverse steps of its removing pattern, we take the inverse of any removal pattern to be an inferred addition if it is not already in the catalog.

These rules are not exhaustive, and other patterns can be inferred by additional rules (e.g., by composing patterns). However, we restrict inference to rules I1 and I2 on the basis that the existence of their inferred patterns is suggested by the reported inverse non-inferred patterns. To differentiate inferred patterns from non-inferred ones, we introduce a fourth label: 'I'.

4 Pattern Catalog

This section presents the resulting catalog of feature evolution patterns. Table 3 lists all the patterns and their usage frequency in our samples. Compared to the previous catalog [52], the patterns herein reported are either as reported before (labelled with an 'O' in the *Info* column of Table 3), a generalization of a pattern in our earlier version (labelled with a 'G'), completely new (labelled with 'N'), or inferred (labelled with an 'I'). In the latter case, a pattern is found in one of the samples, but an inverse pattern does not exist in the other sample, although it is likely to exist in the evolution of the kernel. For instance, if one adds a visible (promptable) feature controlling a specific compilation flag as prescribed by the AVOCFF pattern (row 4), it is also the case that the same feature should be later removed in the course of evolution, although such pattern is not seen in the removals sample. The presented catalog also removes one pattern from our earlier version, as the pattern is not applied by at least three different developers, nor could it be inferred.

We discuss all the patterns in the following, except for rename (RNM), which we omit due to its simplicity.[17] We also present a brief discussion over changes that do not lead to patterns.

4.1 Feature Addition Patterns (Non-Inferred)

Non-inferred patterns are those respecting our two criteria for identifying a pattern, i.e., there exists at least three instances of the change, each from a distinct source of evidence (different developer). We present nine non-inferred patterns in the additions sample concerning two specific situations: (i) adding a new feature from completely new elements (AVOMF, AVOGMF, AVONMF, AVOCFF, AVONMCFF, AVMVF, and AIMF); (ii) adding a new feature created out of existing elements—featurization (FCUTVOF and FCFTVOF). Altogether, they capture how the mapping and implementation change upon adding a new feature in the variability model namespace.

[17] Basically, a rename just updates all references to a given feature name f to a new name f_N in all spaces where f appears. Note that renaming does not cause any behavioural change, nor does it change the set of cross-tree constraints.

	Additions sample			Removals sample		
	Pattern	**Frequency**	**Info**	**Pattern**	**Frequency**	**Info**
1	AVOMF	124	O	RVOMF	22	O
2	AVOGMF	11	O	RVOGMF	12	O
3	AVONMF	32	O	RVONMF	10	O
4	AVOCFF	4	N	RVOCFF	0	I
5	AVONMCFF	3	N	RVONMCFF	0	I
6	AVOAF	2	I	RVOAF	6	N
7	AVMVF	3	N	RVMVF	3	N
8	AIMF	12	O	RIMF	3	O
9	ACINMF	2	I	RCINMF	3	N
10	FCUTVOF	10	G	MVOFNO	3	O
11	FCFTVOF	4	N	MVOFS	3	O
12	RNM	11	O	RNM	18	O
	Total	218		**Total**	83	
	Sample %	81%		**Sample %**	63%	

Table 3: Collected patterns and their frequency

Add Visible Optional Modular Feature (AVOMF). A visible and optional modular feature increases the user configuration space by providing a functionality unit that can be optionally present in the resulting kernel. *Modularity*, in this case, assures that the capabilities of the new feature reside in its own compilation unit(s).

As shown in Figure 9, the pattern adds a new optional and visible feature f in the variability model, along with its associated cross-tree constraints (CTC_f). A build rule then relates the feature presence to its compilation units, whose files are added to the implementation space. The addition of CAPTURE_DAVINCI_DM646X_EVM, previously discussed in Section 3, is an instance of this pattern.

Most primary features in the additions sample (46%) fit into this pattern. To verify where the instances of this pattern add features to, we slice the kernel according to seven subsystems, namely *arch*, *core*, *driver*, *firmware*, *fs*, *misc*, and *net*. Such slicing was proposed by Greg Kroah-Hartman, one of the main kernel maintainers, when collecting different statistics of the evolution of the Linux kernel [13]. These subsystems consist of files from different directories of the kernel source code tree. The code tree is organized in 21 top-level folders, whose descriptions are given in Table 4. A mapping between the kernel source code tree to its associated subsystems is summarized in Table 5, with a bullet indicating that at least one file in a given folder (row) maps to the corresponding subsystem (column). The complete map is publicly available in Hartman's GitHub repository.[18] By applying Hartman's mapping to each Kconfig file, we take the subsystem of a feature to be the same of its enclosing Kconfig file. Once we associate each feature with a single subsystem, we count the number of pattern instances adding primary features to each kernel subsystem (see

[18] `https://raw.github.com/gregkh/kernel-history/master/scripts/genstat.pl`

Table 6). In the case of the AVOMF pattern, its instances add features to the following subsystems:

- *Device driver* (*driver*): 93.6% of the instances in this pattern concern the addition of device drivers (i.e., features that are "plugged-in" to the kernel to support different hardware). This high frequency is in line with previous work [21,23,27,40] stating that Linux kernel evolution is mainly driven by the addition of new device driver-related features.
- *Architecture specific code* (*arch*): 2.4% of the instances of this pattern add modules that are specific to a given hardware architecture. For example, one instance adds support for injecting machine checks when testing the kernel for the x86 architecture. Such functionality is used by kernel developers when performing quality assurance.
- *File system* (*fs*): 1.6% of AVOMF features relate to adding file system functionalities, including support for integrity tests and compression support (LZO) for the Squash file system.[19]
- *Network* (net): 1.6% of the features of this pattern provide network capabilities, such as extending a network protocol with a new functionality. One specific case adds probing support for incoming SCTP packets.
- *Core functionality* (*core*): 0.8% of the features of this pattern add a module to the core subsystem. An example is self-test for 64-bit atomic instructions.

Instances of this pattern are either tristate (91%) or Boolean. The dominance of tristate features follows a trend of most of the patterns related to modular features, evidencing a strong relationship between the two. This association is unlikely to be accidental, as modular tristate features provide flexibility to cover different requirements and configuration purposes. For example, in embedded platforms where hardware can be anticipated, tristate features can be statically linked against the final kernel; in other situations, when hardware configuration varies, tristate features can be compiled as modules and loaded as needed.

It is worth noting that a modular feature can still be referenced in code extensions (*ifdefs*) elsewhere. In such cases, the feature is scattered across files that are not the compilation units of the feature. To verify the number of AVOMF primary features scattered across the Linux kernel code, we iterate over each AVOMF instance, checking out the stable kernel release that adds the primary feature under analysis. Then, we collect all *ifdefs* in code whose condition refers to the name of the primary feature. Such strategy allows to overcome the scope limitation imposed by the commit window size.

We find that only 13 (10%) of the primary features of this pattern are scattered elsewhere, with a small number of *ifdefs*. The fact that the majority of features of this pattern concern modular drivers that cause little scattering suggests that adding driver features aligns with the kernel's architecture, as their modules are "plugged-in" to the system, registering themselves as handlers to specific events (e.g., hardware interrupts) [14]. Scattered modular drivers

[19] `http://squashfs.sourceforge.net/`

Folder	Description
arch	Architecture (CPU) dependent code
block	I/O scheduling algorithms for block devices
crypto	Cryptography related-algorithms
Documentation	Brief descriptions of each part of the implemented kernel
drivers	Device drivers of different devices classes
firmware	Device firmware needed to use certain drivers
fs	Defines the virtual file system abstraction, along with concrete file systems
include	Kernel header files
init	Kernel boot and initialization
ipc	Support for inter-process communication (IPC)
kernel	The main kernel code (architecture independent)
lib	Library (helper) routines
mm	Memory management support
net	Network protocols implementation
samples	Different code examples
scripts	Different scripts for building the kernel
security	The security framework of the kernel, known as LSM (Linux Security Modules), supporting different access control models [69]
sound	The Linux sound subsystem and related device drivers
usr	Implementation of initramfs, a RAM-based root filesystem required by the startup process; the first process (init) runs on top of it
tools	Tools for building the kernel and helper programs useful for kernel developers
virt	Virtualization support

Table 4: Description of the top-level folders of the Linux kernel source code tree (based on [11,41,67])

account for 11 cases in total, with a median number of one *ifdef* (min=1, max=6). Of these 11 drivers, most (6) are scattered across files in the *driver* subsystem; the remaining (5) are scattered across other subsystems, with extensions in *arch* (4) and *core* (1). The two other scattered features are located in *fs*. Different from the drivers' case, their scattering is completely restricted to *fs*, introducing two and six *ifdefs*, respectively. It is not surprising that the scattering in *fs* is local to this subsystem, as the *Virtual File System* in the kernel acts as an abstraction layer for any specific file system and its supported feature set.

Add Visible Optional Guard Modular Feature (AVOGMF). This pattern is a specialization of AVOMF. However, we distinguish between the two and count them separately because the structure of AVOGMF plays an important role in the compilation process. In addition to the changes imposed by AVOMF, the AVOGMF pattern requires that f acts as a compilation guard over an entire directory, controlling whether the compilation process should recursively descend to that location. As such, it contains an additional mapping rule in the parent Makefile:

Source code folder	Subsystems						
	arch	**core**	**driver**	**firmware**	**fs**	**misc**	**net**
arch	•						
block		•					
crypto			•				
Documentation						•	
drivers			•				
firmware				•			
fs					•		
include	•	•	•			•	•
init		•					
ipc		•					
kernel		•					
lib		•					
mm		•					
net							•
samples						•	
scripts						•	
security			•				
sound			•				
usr						•	
tools						•	
virt		•					

Table 5: Mapping of the kernel's top-level directories and its subsystems

Pattern	Distribution across subsystems						
	arch	**core**	**driver**	**firmware**	**fs**	**misc**	**net**
AVOMF	3	1	116	0	2	0	2
AVOGMF	0	0	9	0	0	0	2
AVONMF	7	2	19	0	3	0	1
AVOCFF	0	2	2	0	0	0	0
AVONMCFF	2	0	1	0	0	0	0
AVMVF	0	1	2	0	0	0	0
AIMF	0	0	11	0	0	0	1
FCUTVOF	0	0	10	0	0	0	0
FCFTVOF	0	1	3	0	0	0	0
RNM	0	0	10	0	1	0	0

Table 6: Frequency of non-inferred patterns per subsystem (additions sample)

$$M' = \langle \ldots \overbrace{(f, f/)}^{\textit{in parent Makefile}} \ldots \underbrace{(f, f.o)}_{\textit{in child Makefile (inside f/)}} \ldots \rangle$$

This rule instructs Kbuild to enter a child directory f upon the presence of that feature. Once Kbuild enters the f folder, it processes a Makefile with the rule on how to build f itself. Note that the condition over $f.o$ in the build rule in the child Makefile is redundant. Developers, however, tend to include it to prevent others from interpreting that the compilation of $f.o$ is

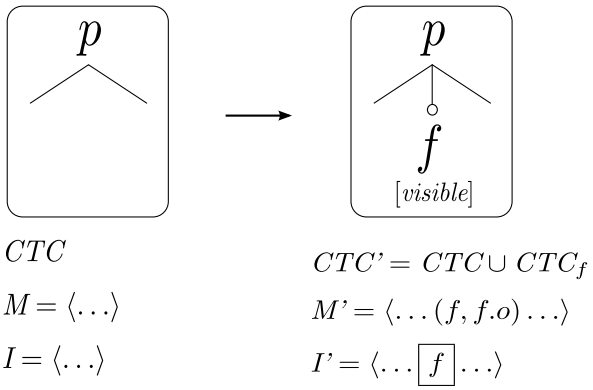


Fig. 9: Definition of Add Visible Optional Modular Feature (AVOMF)

drivers/net/wireless/rtlwifi/Makefile

```
+obj-$(CONFIG_RTL8192SE) += rtl8192se/
...
```

drivers/net/wireless/rtlwifi/rtl8192se/Makefile

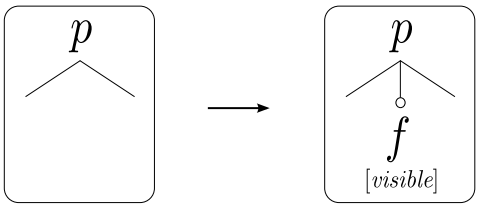
```
+rtl8192se-objs := dm.o fw.o hw.o led.o phy.o rf.o \
+                  sw.o table.o trx.o
+
+obj-$(CONFIG_RTL8192SE) += rtl8192se.o
...
```

Fig. 10: Example of Add Visible Optional Guard Modular Feature (AVOGMF)

not subject to the presence of the f feature. The addition of the device driver supporting Realtek's© 8192 network adapter illustrates this (see Figure 10):[20] in the parent Makefile (top snippet in the figure), Kbuild assesses whether RTL8192SE is present. If so, it enters the rtl8192se directory and processes the child Makefile there (bottom snippet); in that case, RTL8192SE's presence enables the compilation of all objects in the rtl8192se-objs list.

This pattern comprises 4% of all additions, and two idioms result from its usage: (a) developers create guard modular features to control the compilation of a single feature, whose implementation is given by the files in the guarded directory. This represents 82% of the instances of this pattern, where all instances add features to the *driver* subsystem; (b) a guard modular feature roots a subtree in the variability model with, at least, one modular descendant

[20] See http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=85e09b40



CTC $\quad$ $CTC' = CTC \cup CTC_f$

M $\quad$ M

$I = \langle \ldots \rangle$ $\quad$ $I' = \langle \ldots (\ldots f \ldots, C_1, C_2) \ldots \rangle$

Fig. 11: Definition of Add Visible Optional Non-Modular Feature (AVONMF)

feature. All modular features in the subtree reside in the f directory. All the instances of the AVOGMF pattern that relate to this idiom usage add features to the *net* subsystem.

Add Visible Optional Non-Modular Feature (AVONMF). This pattern concerns the addition of features that do not fit inside a module, but rather reside in an existing host code; 12% of the additions instances match this pattern.

As shown in Figure 11, this pattern adds a visible optional feature in the variability model, while not changing the mapping. The implementation changes by including new conditionally compiled code blocks whose condition refers to f (note that the alternative code C_2 may be absent).

This pattern serves the purpose of extending existing capabilities in code. The following patch snippet illustrates this:[21]

```
+#ifdef CONFIG_SQUASHFS_4K_DEVBLK_SIZE
+#define SQUASHFS_DEVBLK_SIZE 4096
+#else
+#define SQUASHFS_DEVBLK_SIZE 1024
+#endif
```

If SQUASHFS_4K_DEVBLK_SIZE (matches f) is present, the block size of the Squash file system is set to four kilobytes; otherwise it is set to one kilobyte.

Following the granularity measures proposed by previous studies [32,38], we verify at which granularity level these extensions take place. The granularity level is defined by the smallest enclosing context, with seven possible levels: *global* (e.g., an *ifdef* annotating an entire function declaration), *function* (e.g., an *ifdef* annotating a statement in the body of a function), *type* (e.g., an *ifdef* annotating a field in a struct), *block* (e.g., an *ifdef* annotating a statement inside the body of a for-loop), *statement* (e.g., an *ifdef* annotating the type or a name of a variable declaration), *expression* (e.g., an *ifdef* annotating

[21] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=7657cacf`

the use of a particular operator or operand), and *function signature* (e.g., an *ifdef* annotating a function parameter declaration). Coarse-grained extensions control the inclusion/exclusion of entire functions or data structures, whereas fine-grained extensions control source code pieces, such as statement and expression extensions or function signature changes [32]. From a total of 122 code *ifdefs* in the commit windows of all non-modular features of this pattern, 44.3% are extensions at the global level (e.g., declaring a new macro, variable, function, structure, etc.), 32.8% occur at the function level (e.g., by adding statements inside a function), 13.1% extend a block statement (e.g., adding a statement inside an if-block), and 9% extend a type declaration (e.g., adding a field to a structure). This distribution is similar to the one found by Liebig et al. [38] when investigating 40 pre-processor-based systems. As we found only a single case (0.8%) of an extension at the statement level and no extensions at the level of expressions or function signatures, our findings strengthens the claim of Liebig et al. that fine-grained extensions are not frequent in practice. Interestingly, f negatively affects the conditionally compiled code in 3% of the extensions, i.,e., its presence excludes a portion of code in the post-processed file (negated f guards an *ifdef* block that does not have an else part).

In contrast to the modular features, in 94% of the instances of this pattern, f is a Boolean feature. Since it does not introduce any compilation unit (and thus, no build rules), it is not possible to directly control whether f should be statically present in the resulting kernel or whether it should be possible to load f dynamically at runtime. The only situation in which f is tristate is when it contains a reverse dependency to a modular tristate feature f_s; if declared as Boolean, f would cause f_s to be statically compiled into the resulting kernel, and thus, breaking the flexibility of the runtime variability related to f_s. However, visible optional non-modular tristate features are rather infrequent, as only two instances appear in our sample; one of them has no selection towards another tristate feature, and thus, provides no benefit over a Boolean declaration.

Most instances of the AVONMF pattern add features to the *driver* subsystem (59.3%), although less frequently than AVOMF instances. In the remaining, 21.9% relate to adding features in *arch*, 9.4% in *fs*, 6.3% in *core*, and 3.1% in *net*.

Add Visible Optional Compilation Flag Feature (AVOCFF). This pattern captures the addition of features that exist with the sole purpose of enabling specific compilation flags; it comprises 1% of all additions in our sample. This pattern is new in our catalog and follows directly from our refined recurrence measure. The purpose of the pattern is to expose a compilation flag that enables specific diagnostic capabilities, such as profiling and debug messages. Figure 12 shows the pattern, and an example is given in Figure 13.[22] Selecting USB_DWC3_VERBOSE, a new feature added to the Kconfig model (Figure

[22] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=72246da4`

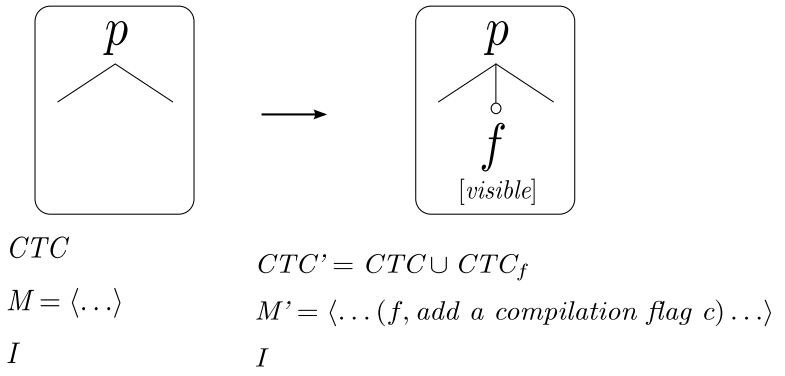


Fig. 12: Definition of Add Visible Optional Compilation Flag Feature (AVOCFF)

```
1  drivers/usb/dwc3/Kconfig
2
3  +config USB_DWC3_VERBOSE
4  +   bool "Enable Verbose Debugging Messages"
5  +   depends on USB_DWC3_DEBUG
6  +   help
7  +      Say Y here to enable verbose debugging messages on
8  +      DWC3 Driver.
9  +
10 ...
11
12 drivers/usb/dwc3/Makefile
13
14 +ccflags-$(CONFIG_USB_DWC3_VERBOSE) += -DVERBOSE_DEBUG
15 ...
16
```

Fig. 13: Example of Add Visible Optional Compilation Flag Feature (AVOCFF)

13, lines 3–8), defines the macro symbol VERBOSE_DEBUG, which is then referred in code, controlling whether calls to specific debug routines should be in the post-processed file. The definition of VERBOSE_DEBUG occurs by adding the compilation flag -DVERBOSE_DEBUG to the C flags list (ccflags).

In the investigated sample, half of the AVOCFF instances add features to *core*, while the remaining add features to *driver*.

Add Visible Optional Non-Modular Compilation Flag Feature (AVONMCFF). This pattern is a composition of AVONMF and AVOCFF. It is not accounted in neither AVONMF nor AVOCFF, as the former does not change the mapping, whereas the latter does not affect the implementation. To cover both types of

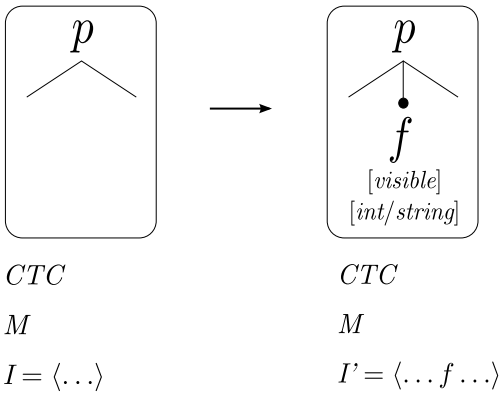


Fig. 14: Definition of Add Visible Mandatory Value-Based Feature (AVMVF)

changes, we introduce the new pattern AVONMCFF, which is equivalent to the composition of the two base patterns. The result of the AVONMCFF pattern in the after state is a new visible optional feature in the variability model, and a new compilation flag whose activation is subject to the presence of the newly added feature, together with *ifdefs* in code that refer to it. Since the new feature does not hold a compilation unit of its own, it is non-modular. The pattern has three instances, corresponding to 1% of the additions sample size. All three instances are Boolean, adding features to *arch* (2) and *driver* (1).

Add Visible Mandatory Value-Based Feature (AVMVF). This pattern, shown in Figure 14, covers the addition of a mandatory visible value-based feature (integer or string). As the feature is just a place-holder for a value, it does not add any cross-tree constraint, nor any compilation unit, preserving both *CTC* and *M*. The feature is, however, referred in the implementation when initializing specific parts of the code. Figure 15 exemplifies this.[23] The newly added value-based feature RCU_BOOST_PRIO is referred in kernel/rcutiny.c (line 27) to initialize a scheduling parameter. Three instances of our sample (1%) fall into this pattern, adding features to *core* (1) and *driver* (2).

Add Internal Modular Feature (AIMF). Internal modular features are not directly exposed to users during configuration, as they are invisible (non-promptable). Such features exist to provide a common infrastructure to other features, which in turn select them by means of reverse dependencies. Overall, this pattern comprises 4% of all additions in our sample.

This pattern describes how internal modular features are added: as with other modular features, the variability model, mapping, and implementation change to accommodate the new feature (referred as f_1). However, two key characteristics arise: (i) f_1 is invisible; (ii) an additional constraint states

[23] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=24278d14`

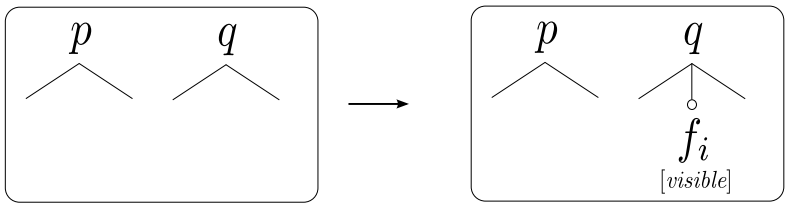
```
init/Kconfig

+config RCU_BOOST_PRIO
+  int "Real-time priority to boost RCU readers to"
+  range 1 99
+  depends on RCU_BOOST
+  default 1
+  help
+     This option specifies the real-time priority to which
+     preempted RCU readers are to be boosted.  If you are
+     working with CPU-bound real-time applications, you
+     should specify a priority higher then the highest-priority
+     CPU-bound application.
+
...

kernel/rcutiny.c

...
 static int __init rcu_spawn_kthreads(void)
 {
-  rcu_cbs_task = kthread_run(rcu_cbs, NULL, "rcu_cbs");
+  struct sched_param sp;

+  rcu_kthread_task = kthread_run(rcu_kthread, NULL,
+                                 "rcu_kthread");
+  sp.sched_priority = RCU_BOOST_PRIO;
+  sched_setscheduler_nocheck(rcu_kthread_task, SCHED_FIFO, &sp);
   return 0;
 }
...
```

Fig. 15: Example of Add Visible Mandatory Value-Based Feature (AVMVF)

that another feature f_2 selects f_1 (represented as an implication). Thus, the cross-tree constraints in the after-state are: $CTC' = CTC \cup CTC_{f_1} \cup \{f_2 \rightarrow f_1\}$.

Except for one feature in *net*, all other instances of AIMF concern the addition of driver-related features (92%).

Featurize Compilation Unit to Visible Optional Feature (FCUTVOF). Featurization occurs when existing elements are exposed as new features. One specific kind of featurization is when an existing compilation unit, initially subject to the presence of a feature p, becomes associated with its own feature, which is in turn created as a result. Such situation occurs in 4% of additions.



CTC

$M = \langle \ldots (p, f_1.o \ldots f_n.o) \ldots \rangle$

$I = \langle \ldots \boxed{f_1} \ldots \boxed{f_n} \ldots \rangle$

$CTC' = CTC \cup CTC_{f_i}$

$M' = \langle \ldots (p, f_1.o \ldots f_{i-1}\ f_{i+1} \ldots f_n.o)$
$\ldots (f_i, f_i.o) \ldots \rangle$

$I = \langle \ldots \boxed{f_1} \ldots \boxed{f_n} \ldots \rangle$
with $1 \leq i \leq n$

Fig. 16: Definition of Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)

In the extracted pattern, illustrated in Figure 16, a feature p controls a set of object files $f_1.o \ldots f_n.o$. One of these objects, however, is not essential to the functionality provided by p; rather, its capability is optional. In this case, $f_i.o$ is featurized, i.e., a new feature f_i is created to control whether $f_i.o$ should be compiled or not. The new feature, in turn, is placed in the variability model under an existing feature q. Features p and q may or not be the same, which generalizes our original definition in [52],[24] which imposed p and q to be equal. Upon the creation of f_i, $f_i.o$ is then removed from the list of objects controlled by p. Featurizing $f_i.o$ gives users a finer-grained control over the configuration process, while decreasing the granularity of p. That prevents unnecessary functionality to be shipped in the resulting kernel, and in turn, improves its memory usage and boot time. The example shown in Figure 17 illustrates the featurization of me4000.o, previously controlled by COMEDI_PCI_DRIVERS, into the new feature COMEDI_ME4000.[25]

All 10 instances of the FCUTVOF pattern add features to the *driver* subsystem.

Featurize Code Fragment to Visible Optional Feature (FCFTVOF). In this featurization pattern (see Figure 18), an unconditional code fragment C_0 becomes conditionally compiled and bound to the presence of a newly added feature f. To cover the case where f is not present, an alternative piece of code is given (C_1). When C_1 is not empty, the goal of the pattern is to provide an alternative behavior to an already existing implementation. Otherwise, the

[24] In our previous catalog [52], we name the pattern as *Featurize code (FTC)*.

[25] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f1d7dbbe`

drivers/staging/comedi/Kconfig

```
menuconfig COMEDI_PCI_DRIVERS
   tristate "Comedi PCI drivers"

+config COMEDI_ME4000
+   tristate "Meilhaus ME-4000 support"
+   help
+     Enable support for Meilhaus PCI data acquisition cards
+     ME-4650, ME-4670i, ME-4680, ME-4680i and ME-4680
...
```

drivers/staging/comedi/drivers/Makefile

```
-obj-$(CONFIG_COMEDI_PCI_DRIVERS) += me4000.o
+obj-$(CONFIG_COMEDI_ME4000) += me4000.o
...
```

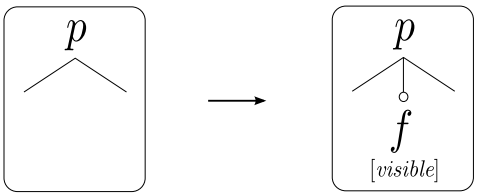
Fig. 17: Example of Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)

pattern extracts optional behaviour, decreasing the footprint of the resulting object code, which improves overall performance.

This FCFTVOF pattern covers 1% (4) of the sampled additions, and for the most part (3) it concerns the featurization of code fragments in driver-related features. Figure 19 provides an example of the featurization of volume-related functions in the subdriver of the ACPI ALSA driver for ThinkPad©.[26] If THINKPAD_ACPI_ALSA_SUPPORT is present (a newly added feature), the volume-subdriver registers support for volume capabilities (not shown) and successfully initializes, as given by the return value in its init function (Figure 19, line 27); otherwise, THINKPAD_ACPI_ALSA_SUPPORT is not present, and volume capability-functions are not compiled in the resulting driver, causing the initialization of the volume-subdriver to fail, as given by the return value one (line 39). The commit log message of the patch confirms that the featurization is motivated by performance optimization:

> *"Allow the user to choose through Kconfig if the Console Audio Control interface (aka "volume subdriver") should be available or not. This not only saves some memory, but also allows the thinkpad-acpi driver to be built-in even if ALSA is modular when the console audio control interface is not wanted..."*

[26] See http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ff850c33



CTC	$CTC' = CTC \cup CTC_f$
M	M
$I = \langle \ldots C_0 \ldots \rangle$	$I' = \langle \ldots (\ldots f \ldots, C_0, C_1) \ldots \rangle$

Fig. 18: Definition of Featurize Code Fragment to Visible Optional Feature (FCFTVOF)

4.2 Feature Addition Patterns (Inferred)

We infer two patterns in the additions sample: *Add Visible Optional Abstract Feature (AVOAF)* and *Add Computed Internal Non-Modular Feature (ACINMF)*. Both inferred patterns are bellow the threshold of three instances, but they have a corresponding inverse non-inferred pattern in the removals sample. The existence of an inverse non-inferred pattern in the removals sample suggests the inferred ones.

Add Visible Optional Abstract Feature (AVOAF). This inferred pattern concerns the addition of *abstract features*, i.e., features that are exclusive to the variability model, and thus, are not referred in other spaces [65]. All the four cases of adding abstract features in the unexcluded portion of the additions sample relate to Boolean and optional features, but only half are visible. Thus, this pattern is under our set threshold, as it has only two instances. However, as we report an inverse non-inferred pattern (RVOAF) in the removals sample (see Section 4.3), we classify these two visible features as part of an inferred pattern in the additions sample.

Interestingly, all abstract features in the addition sample are leafs in the variability model (as opposed to being internal nodes). In the cases where these abstract features are visible, their addition aims at capturing a configuration aspect that other features rely on. These features, in turn, do affect the mapping and/or implementation. Figure 20 illustrates this:[27] the addition of the visible optional abstract feature RD_XZ in the *misc* subsystem (lines 4–12) captures whether users want support for initial RAM disk compression. An initial RAM disk (initrd) is an initial root file system loaded as part of the kernel booting process, providing a minimal set of directories and executables that support the booting process (e.g., the insmod executable will be called to load different kernel modules, such as device drivers) before the actual file system

[27] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=3ebe1243`

```
drivers/platform/x86/Kconfig

+config THINKPAD_ACPI_ALSA_SUPPORT
+   bool "Console audio control ALSA interface"
+   depends on THINKPAD_ACPI
+   depends on SND
+   depends on SND = y || THINKPAD_ACPI = SND
+   default y
+   ---help---
+     Enables monitoring of the built-in console audio output
+     control (headphone and speakers), which is operated by
+     the mute and (in some ThinkPad models) volume hotkeys.
...

drivers/platform/x86/thinkpad_acpi.c

+#ifdef CONFIG_THINKPAD_ACPI_ALSA_SUPPORT
...
// Volume-related functions
...
static int __init volume_init(struct ibm_init_struct *iibm)
 {
    ...
    vdbg_printk(TPACPI_DBG_INIT,
                "initializing volume subdriver\n");
    ...
    return 0;
}
...
+#else /* !CONFIG_THINKPAD_ACPI_ALSA_SUPPORT */
+
+#define alsa_card NULL
+ ...
+static int __init volume_init(struct ibm_init_struct *iibm)
+{
+   printk(TPACPI_INFO,
+          "volume: disabled as there is no ALSA support...\n");
+
+   return 1;
+}
+
+#endif

```

Fig. 19: Example of Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)

```
1
2  usr/Kconfig
3
4  +config RD_XZ
5  +   bool "Support initial ramdisks compressed using XZ"
6  +        if EMBEDDED
7  +   default !EMBEDDED
8  +   depends on BLK_DEV_INITRD
9  +   select DECOMPRESS_XZ
10 +   help
11 +      Support loading of a XZ encoded initial ramdisk or cpio
12 +      buffer. If unsure, say N.
13 +
14 ...
15
16 lib/Makefile
17
18 +# XZ
19  lib-$(CONFIG_DECOMPRESS_LZMA) += decompress_unlzma.o
20 +lib-$(CONFIG_DECOMPRESS_XZ)   += decompress_unxz.o
21 +
22 ...
```

Fig. 20: Example of Add Visible Optional Abstract Feature (AVOAF)

is mounted. An initial RAM disk is kept as a compressed file, which is then uncompressed during the boot and placed in the primary memory (RAM). Upon the selection of RD_XZ, a reverse dependency selects DECOMPRESS_XZ, causing decompress_unxz.o to be compiled in a supporting library for the kernel (line 20). The other instance of this inferred pattern concerns the addition of an IPV4 feature in *net*.

The other two situations of adding abstract features relate to invisible ones. The two invisible features are capability abstractions [9] over the target hardware architecture for the kernel. Figure 21 illustrates this:[28] HAVE_KERNEL_GZIP (line 3) abstracts over gzip compression support of the target kernel image. As this functionality is not specific to x86, another feature KERNEL_GZIP exists, and its selection depends on the existing support of the target hardware architecture. Hence, x86 explicitly states its supported capabilities by selecting them, which includes HAVE_KERNEL_GZIP (line 23). Although this situation is actually prescribed in the Kconfig manual,[29] it was not found recurrent in our sample, and thus, we do not report it as a pattern. Moreover, it cannot be inferred, as we do not report an inverse pattern in the removals sample.

[28] See http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=2e9f3bdd

[29] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

```
init/Kconfig

+config HAVE_KERNEL_GZIP
+  bool
+

 config KERNEL_GZIP
-   bool "Gzip"
+   depends on HAVE_KERNEL_GZIP
    help
      The old and tried gzip compression. Its compression ratio
      is the poorest among the 3 choices; however its speed
      (both compression and decompression) is the fastest.

...

arch/x86/Kconfig

config X86
   select HAVE_GENERIC_DMA_COHERENT if X86_32
   select HAVE_EFFICIENT_UNALIGNED_ACCESS
   select USER_STACKTRACE_SUPPORT
+  select HAVE_KERNEL_GZIP
...
```

Fig. 21: Example of an invisible optional abstract feature

Add Computed Internal Non-Modular Feature (ACINMF). This inferred pattern concerns the addition of a feature that is not promptable, and thus, it is invisible to users. Its presence is computed from a constraint setting the default value of the feature. The added feature is referred in code by means of *ifdefs*; as it does not have a compilation unit, the feature is non-modular. Computed internal features exist to encapsulate specific constraints, which simplifies the encoded variability; instead of repeating the constraint at each variation point that it is needed, developers encapsulate it in a single feature, which facilitates later maintenance when updating the constraint. Two instances of this inferred pattern appear in the additions sample, and both concern Boolean features being added to *arch* and *core*, respectively.

4.3 Feature Removal Patterns (Non-Inferred)

Non-inferred patterns in the removal sample capture how the mapping and implementation spaces change, if at all, upon the removal of an existing feature in the variability model. Excluding rename (RNM), we report nine non-inferred

Pattern	Distribution across subsystems						
	arch	**core**	**driver**	**firmware**	**fs**	**misc**	**net**
RVOMF	1	1	20	0	0	0	0
RVOGMF	0	0	12	0	0	0	0
RVONMF	0	0	10	0	0	0	0
RVOAF	0	0	4	0	0	0	2
RVMVF	0	0	2	0	0	0	1
RIMF	0	0	3	0	0	0	0
RCINMF	2	0	1	0	0	0	0
MVOFNO	0	0	3	0	0	0	0
MVOFS	0	0	3	0	0	0	0
RNM	0	0	16	0	1	0	1

Table 7: Frequency of non-inferred patterns per subsystem (removals sample)

patterns in the removals sample, from which seven capture retirement situations directly matching their counterpart in the additions sample: *Retire Visible Optional Modular Feature (RVOMF)*, *Retire Visible Optional Guard Modular Feature (RVOGMF)*, *Retire Visible Optional Non-Modular Feature (RVONMF)*, *Retire Visible Optional Abstract Feature (RVOAF)*, *Retire Visible Mandatory Value-Based Feature (RVMVF)*, *Retire Internal Modular Feature (RIMF)*, and *Retire Computed Internal Non-Modular Feature (RCINMF)*. Among these, retirement patterns removing visible optional features and affecting the implementation space account for most removal cases. The inverse addition patterns matching these removal patterns show the same trend. Thus, both trends suggest that the kernel evolution is mainly driven by adding or removing visible optional features with some associated implementation. Moreover, as observed in the additions sample, most remove patterns relate to features in the *driver* subsystem (see Table 7).

Kernel maintainers retire features when: (a) the features are under staging (unstable features) for a long time, and there is no indication that they will gain enough quality to be merged into the main kernel. Reasons include broken, unmaintained, or buggy features, or non-adherence to development conventions; (b) the features break due to changes elsewhere and no effort is put to fixing them; (c) the features are not used and are unmaintained for a long time; (d) another feature supersedes an obsolete one, causing the latter to be retired.

Interestingly, 67% of RIMF and RVMVF, 64% of RVONMF, 50% of RVOAF, and 27% of the RVOMF instances are removed as a consequence of retiring the whole subtree containing them. This suggests that some forms of retirement occur in a coarse-grained manner and are triggered by the removal of a feature rooting an entire subtree, along with all its descendants.

Non-retirement patterns also exist, and capture cases where a feature is merged into another one. Two such patterns exist: *Merge Visible Optional Feature into New One (MVOFNO)* and *Merge Visible Optional Feature into Sibling (MVOFS)*. The instances of each merge pattern concern the merging of features in the *driver* subsystem. It is worth noting that in our earlier catalog [52], we also reported a third pattern: *Merge Visible Optional Feature into*

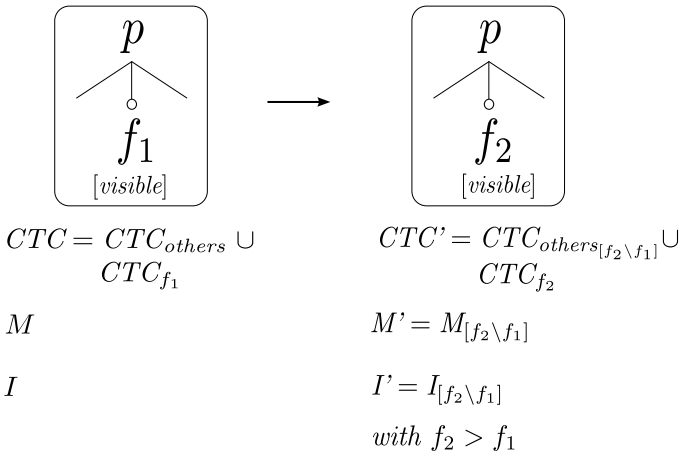


Fig. 22: Definition of Merge Visible Optional Feature into New One (MVOFNO)

Computed Internal. Such pattern, however, has been dropped from this new catalog, since it does not contain at least three distinct sources of evidence, nor could it be inferred. We present MVOFNO and MVOFS in the following.

Merge Visible Optional Feature into New One (MVOFNO). This pattern concerns the creation of a feature from an existing one, which is then enhanced with new code. Figure 22 illustrates the pattern. A feature f_1 is renamed to f_2, and its set of cross-tree constraints is replaced with a new set CTC_{f_2}. Furthermore, all references to f_1 are replaced by references to f_2 in all spaces. At the implementation level, $f_2 > f_1$ captures the enhanced code, meaning that f_2 supports all the capabilities of f_1, plus new ones.

Of all instances in the removals sample, 2% (3) fit into this pattern and often relate to generalizing drivers to support a set of related hardware family.

As a concrete example, consider the merge of BATTERY_PALMTX into the new feature BATTERY_WM97XX supporting a whole family of chips.[30] As shown in the associated patch (see Figure 23), developers drop the original cross-tree constraints and rename the previous feature from the variability model and mapping. Moreover, the code is updated with various information about the new driver (not shown). Note that in the example, the merge changes the associated help text, but it does not relate the new feature back to BATTERY_PALMTX. Thus, when users migrate towards a newer kernel with BATTERY_WM97XX, they may incorrectly conclude that BATTERY_PALMTX is no longer supported. Hence, merges can cause the false impression that some features cease to exist.

Merge Visible Optional Feature into Sibling (MVOFS). This pattern covers the situation in which developers merge a visible optional feature into its sibling

[30] See `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4e9687d9`

drivers/power/Kconfig

```
-config BATTERY_PALMTX
-   tristate "Palm T|X battery"
-   depends on MACH_PALMTX
+config BATTERY_WM97XX
+   bool "WM97xx generic battery driver"
+   depends on TOUCHSCREEN_WM97XX
    help
-     Say Y to enable support for the battery in Palm T|X.
+     Say Y to enable support for battery measured by WM97xx
...
```

drivers/power/Makefile

```
-obj-$(CONFIG_BATTERY_PALMTX) += palmtx_battery.o
+obj-$(CONFIG_BATTERY_WM97XX) += wm97xx_battery.o
...
```

Fig. 23: Example of Merge Visible Optional Feature into New One (MVOFNO)

(see Figure 2), due to their similarity. The merging of FB_IMAC into FB_EFI, previously discussed in Section 2.2, exemplifies the pattern.

This pattern aims at easing maintenance, as keeping two similar features might require a duplicate effort whenever a change occurs in either of them. As other merges, this pattern is responsible for 2% (3) of all removals in the sample.

4.4 Feature Removal Patterns (Inferred)

We infer two removal patterns: *Retire Visible Optional Compilation Flag Feature (RVOCFF)* and *Retire Visible Optional Non-Modular Compilation Flag Feature (RVONMCFF)*. Opposed to the inferred pattern in the additions sample, we do not find any instances of these two patterns. However, as these feature types are added as seen in the additions sample, it is reasonable to assume that one way of retiring such features is by performing the opposite steps of their addition.

4.5 Non-Patterns in the Additions and Removals Samples

The patterns reported in Table 3 cover most of the additions (81%) and removals (63%) we analyzed. However, not every change results in a pattern. Following the names defined in the previous section, the additions that do not match a pattern and are not excluded from analysis (38) fall into the following cases:

- Addition of guard features (5), i.e., features whose sole purpose is to guarantee the compilation of the content inside a given folder. Although such case respects the defined threshold, this cluster does not hold three distinct sources of evidence.
- Addition of internal optional abstract features (2).
- Addition of computed internal modular features (2).
- Addition of a computed internal non-modular feature whose extension combines new code fragments with existing lines of code (1).
- Addition of an internal mandatory modular feature (1).
- Addition of internal mandatory non-modular features (2).
- Addition of internal optional non-modular features (2).
- Addition of a computed value-based feature, i.e., a value-based feature whose presence is computed (1).
- Different situations of exposing existing code as a feature (15).
- Distinct merge cases (4).
- Featurization of existing constraints in the variability model (2).
- Combination of the rename of a feature and the split of its compilation unit (1).

In the case of the 38 unexcluded instances in the removals sample that are not put as part of a pattern, we report the following situations:

- Removal of individual cases of internal features (4) not fitting RIMF nor RCINMF. A concrete example includes the removal of an internal optional compilation flag feature.
- Different cases where a feature becomes an integral part of the code, while being removed from the variability model (16).
- Different merge situations that do not lead to patterns (17).
- One split case.

Compared to the additions sample, removals tend to contain more merge-related changes, with a rich realization that leads to different ways on how to accomplish them. Consequently, few merge patterns arise.

5 Summary of Findings and Further Discussion

Based on the extracted catalog, we discuss some evolution principles revealed by the analysis of the reported patterns, followed by a discussion of how our patterns already point to deficiencies in state-of-the-art tools/techniques. In addition, we argue for a new evolution theory in the software product line field.

5.1 Kernel Evolution Principles

In our catalog, the two most frequent patterns are AVOMF and AVONMF, accounting for 58% of all investigated feature additions. Together, these two patterns reveal some key principles governing the kernel evolution.

The AVOMF pattern, the most recurrent pattern in our catalog, shows that most additions introduce modular features, i.e., features that have their own compilation unit(s). This high degree of modularity allows the kernel developers to confine implementation under well-defined interfaces (e.g., the driver-development API), causing changes to be localized and fostering parallel development—a key strategy in the distributed settings in which the kernel is developed. If features are not fully modular, they are at least not heavily scattered across the kernel. Instead, scattering is restricted mostly to files in the same subsystem as their associated primary features. This suggests that Linux kernel evolution is kept in line with the underlying software architecture.

The extensions introduced by non-modular features, as prescribed by AVONMF, are coarse-grained, occurring mostly at the global and function levels. Coarse-grained extensions suggest a *disciplined* usage of *ifdefs*, as annotations align with the syntactic units of the host programming language. As argued by Liebig et. al [38,39], disciplined annotations facilitate maintenance activities (e.g., refactoring in the presence of *ifdefs*) and even make it possible to rewrite scattered features by means of other alternative techniques that could modularize them (e.g., using aspects [34]).

The dominance of modular features, low scattering, and coarse-grain annotations mitigates the challenges imposed by the use of *ifdef* annotations on program comprehension [20,30,37,61] and on the potential of introducing bugs [19,33]. While modularity is supported by the plugin architecture of the kernel, low scattering and coarse grain annotations appear to follow directly from coding guidelines related to *ifdef* use:[31]

> *"Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define 'static inline' functions, or macros, which are used in the code. Let the compiler optimize away the "no-op" case."*

The kernel development process also reinforces that understanding:[32]

> *"The C pre-processor seems to present a powerful temptation to some C programmers, who see it as a way to efficiently encode a great deal of flexibility into a source file. But the pre-processor is not C, and heavy use of it results in code which is much harder for others to read and harder for the compiler to check for correctness. Heavy pre-processor use is almost always a sign of code which needs some cleanup work [...] Conditional compilation with #ifdef is, indeed, a powerful feature, and it is used within the kernel. But there is little desire to see code which is sprinkled liberally with #ifdef blocks."*

which is further stressed by Linus Torvalds himself when rejecting a contributed patch:[33]

[31] `https://www.kernel.org/doc/Documentation/SubmittingPatches`

[32] `https://www.kernel.org/doc/Documentation/development-process/4.Coding`

[33] `http://yarchive.net/comp/linux/ifdefs.html`

> *"Note that there is no way I will ever apply this particular patch for a very simple reason: #ifdef's in code [...] And make your #ifdef's be _outside_ the code. I hate code that has #ifdef's. It's a major design mistake [...] So please spend some time cleaning it up, I can't look at it like this."*
>
> —Linus Torvalds, Wed, 8 Aug 2001 09:40:07 (fa.linux.kernel newsgroup)
>
> *Having the #ifdef's outside the code tends to have two advantages:*
>
> *- it makes the code much more readable, and doesn't split things up.*
>
> *- you have to choose your abstraction interfaces more carefully, which in turn tends to make for better code.*
>
> *Abstraction is nice - _especially_ when you have a compiler that sees through the abstraction and can generate code as if it wasn't there.*
>
> —Linus Torvalds, Wed, 8 Aug 2001 12:14:32 (fa.linux.kernel newsgroup)

5.2 Patterns: Empirical Evidence

Deficiency in Existing Tools/Techniques Our pattern catalog lists additions and removal situations that stem from a large and complex real-world system. Although we cannot claim that our patterns are representative of all the changes performed in the evolution of all kinds of variant-rich systems, some patterns already capture real evolution scenarios that some state-of-the-art variability evolution techniques are not able to handle correctly.

To illustrate our point, consider the edit-based reasoning technique proposed by Thüm et al. [64]. They categorize changes in the variability model as:

- *Generalization*: The introduced changes in the variability model do not impact previous valid configurations. The changes, however, allow new valid configurations.
- *Specialization*: The changes decrease the set of previously valid configurations.
- *Refactoring*: The set of valid configurations resulting from the changes remains the same.
- *Arbitrary edit*: None of the above.

In Thüm's approach, reasoning is performed by efficiently translating both the original variability model and the one resulting from the changes into a satisfiability problem; by avoiding an exponential explosion of CNF clauses, the proposed reasoning has been tested over large models, showing to scale with randomly-generated models with up to 10,000 features. Moreover, reasoning does not require variability models to have the same set of features, as generalization can include new ones, and specialization remove others. This is in contrast to previous work [28,62], which limited the focus to either equivalence [62] or specialization [28] of variability models with the same set of features.

Despite the advances of the work of Thüm et al., their approach may not produce sound results in the case of changes that affect the feature set, but that preserve the overall functionality of the target software through changes in other spaces. The merging of FB_IMAC into FB_EFI, discussed in Section 2.2, illustrates this situation. While FB_IMAC is removed from the variability model, FB_EFI supersedes the removed feature in the implementation space. Furthermore, since FB_EFI has the same cross-tree constraints as FB_IMAC, no constraint is lost (a renaming refactoring updates references to FB_IMAC to become references to FB_EFI). This way, functionality is preserved, as support for FB_IMAC is now given by FB_EFI. However, since the edit-reasoning technique of Thüm et al. considers only changes of the variability model, it would report the discussed merge as specialization, which would be incorrect; after the merge, the resulting system would still be compatible with the one prior to the change, and as such, it would preserve all the existing variants. Other techniques, such as those proposed in [28,62], are not even able to process such a change, even if restricted to changes in the variability model only; these techniques require the same feature set.

Finally, patterns provide preliminary evidence of specific evolution practices that ought to be of interest to tool builders. When retiring features, for instance, Linux kernel developers often remove entire subtrees in the variability model, removing all features therein, along with their associated artifacts. Further studies shall confirm whether such practice is also found in other systems.

The Need for New Theories Our catalog shows that feature retirement comprises most of the patterns in the removals sample. Thus, removals are frequent in the evolution of the Linux kernel. While the MVOFS and MVOFNO patterns are captured by the existing theory of software-product-line refinement [10], retirement patterns are not. Thus, a new theory of product-line evolution that covers not only refinement, but also retirement situations is needed. As our catalog is the first of its kind, our patterns can serve as a starting point for understanding specific types of feature removals that should be accounted for in new theories.

6 Threats to Validity

There is a threat that our analysis does not reflect the whole population of feature additions/removals in the Linux kernel. To mitigate this threat, we rely on randomly collected samples in the hope that they are representative of the additions and removals found in the target population of the x86 architecture.

Our scoping decisions threaten external validity. First, our analysis focuses on additions and removals in the variability model of the x86 architecture, while observing how related artifacts coevolve as a result. Despite existing evidence that the variability model of the x86 architecture follows a similar growth in comparison to the variability model of the whole kernel [40], it is not safe to claim that our patterns are representative for all kernel architectures. Similarly,

we cannot claim that our patterns are representative of feature additions and removals as found in other variant-rich software systems, open-source or not. As a first study of its kind, our work shall be succeeded by other studies to verify whether the reported patterns are exclusive to the evolution of the Linux kernel or whether they are also found in other systems. Any system organized in terms of a variability model, a mapping, and source code that relies on *ifdef* annotations, is a prospective candidate. Examples include other open-source Kconfig-based software systems [9,44], the eCos real-time operating system [43], and even industrial product product lines [6].

The size of the samples is a threat in our study, although minor: if a different and larger sample is used, new patterns may be found, possibly with a different frequency. However, the patterns we report are still valid (although not possibly found in the other sample), as they are selected from clusters with at least three instances and with three or more distinct contributors.

The choice of a recurrence measure is a threat to internal validity. We argue, however, that the use of at least three instances assures the inclusion of less frequent patterns, while still requiring a minimal recurrence degree. Furthermore, having at least three instances prevents us from incorrectly reporting non-inferred patterns over extreme outliers (rare evolution scenarios). To avoid bias towards personal change styles, the non-inferred patterns are also required to have three distinct sources of evidence, meaning that the patterns have been employed by, at least, three distinct developers. Inferred patterns, in contrast, do not guarantee the existence of, at least, three instances, nor three distinct sources of evidence. Thus, inferred patterns impose an additional threat. We argue, however, that it is logical to assume the existence of an inferred pattern, as long as we provide evidence that its inverse pattern is not inferred. Such guarantee follows from our methodology (see Section 3). To prevent readers from interpreting inferred patterns as non-inferred ones, we clearly label them with 'I' in Table 3.

Manually extracting and classifying patterns raises a threat to construct validity. We mitigate this threat by devising and following a methodology with a well-defined sequence of steps. Some steps, however, involve subjective analysis (e.g., defining the size of commit windows and cluster categories). Following best practices in case study research [56], subjectivity is mitigated by performing, at least, three extensive reviews of our analysis to guarantee the consistency among all reported patterns. We also document all the collected data and its analyses, making them publicly available for independent verification (see Section 3).

Last, but not least, we acknowledge that our patterns result from an indirect observation of what developers do. As such, despite the fact that we are able to explain most of the additions and removals in our samples, our catalog may not represent the evolution at the same abstraction level as perceived by kernel developers. Moreover, as our patterns directly follow from the analysis of the kernel commit history, they cannot capture any kernel variability evolution practices occurring outside the kernel source code repository.

7 Related Work

Our previous investigation [52] presented a catalog of 13 variability-evolution patterns. In this work, we reanalyze our initial dataset, augmenting it with 30% more additions and removals, and report a new catalog with seven new variability-evolution patterns, four inferred ones (situations that follow from our set of patterns, but that are not seen in the collected sample), and a generalization of a previously reported pattern. While adding new patterns, we also remove one from our earlier catalog, as it does not meet our requirement of at least three distinct sources of evidence.

Although we are the first to consider variability in Linux kernel from the viewpoint of the evolution of its variability model with other related software artifacts, different researchers have studied Linux from other perspectives.

She et al. [58] propose the Linux variability model as a realistic benchmark for evaluating variability modeling tools. By analyzing various metrics (e.g., branch factor, cross-tree constraint ratio, depth, etc.), the authors show that, for the most part, Linux Kconfig models surpass the complexity of models found in the research community.

Lotufo et al. [40] extend She's work with a longitudinal analysis over Linux Kconfig models, in addition to presenting evolution scenarios and operations faced by developers when evolving those models. For the most part, the authors restrict their analysis to variability models, which, as we argued before, leads to an incomplete and possibly misleading understanding of the evolution in place.

Researchers also investigate the problems resulting from the coevolution of the spaces of the Linux kernel. Tartler et al. [63] detect inconsistencies between the variability model and the C code (e.g., an *ifdef* whose condition cannot be satisfied given the set of cross-tree constraints). Nadi et al. [46] extend that framework to detect inconsistencies among different spaces (e.g., a build rule is dead due to an inconsistency with the constraints in the variability model).

Others cover evolution in a multi-space setting, but restrict analysis to small software product lines. Holdschick [26] presents change operations between variability models and functional models in the automotive domain. Neves et al. [48] extract operations conforming to the refinement theory in [10]. Their operations guarantee that old variants can still be mapped to variants in the product line resulting from an operation execution. In contrast, our catalog has no such focus, and further shows that the Linux kernel drops support for specific products during its evolution, as feature retirement often happens.

Seidl et al. [57] present a set of evolution scenarios and mapping operators to reestablish the correct binding of different spaces in a software product line. In contrast to our work, they do not provide any empirical evidence over the need of supporting those scenarios. Furthermore, the authors state that changes are driven either by edits in the variability model or in the implementation side. However, as the FCUTVOF pattern shows, this does not hold entirely, as changes can also stem from the mapping.

Kim et al. [35] propose a rule-based program differencing approach that discovers and summarizes systematic code changes as logic rules. They also use the version control history to detect evolution patterns, as we did. However, they inspect only code differences, whereas we investigate the coevolution of the variability model, Makefiles, and source code.

8 Conclusion

In variant-rich software systems, variability is not restricted to variability models, but it is rather pervasive to different artifacts, such as build files and code. In such settings, variability evolution requires variability models to coevolve with related artifacts. Surprisingly, little is known about such coevolution, with a direct impact in the quality of existing tools.

Attempting to mitigate this overall lack of knowledge, we analyze coevolution in the context of a large and complex case study: The Linux kernel. In particular, we investigate the coevolution of the Linux kernel variability model, Makefiles, and C source code by analyzing a sample spanning almost four years of Linux kernel evolution history. From our investigation, we collect a catalog of evolution patterns that extends our earlier work [52], capturing patterns that were not reported before.

Each pattern in our catalog explains how certain kinds of changes affect the artifact types in the kernel, the frequency of such changes, and how they are used by Linux kernel developers. To the best of our knowledge, our catalog is the first to recover the coevolution of variability models and related artifacts in a large and complex real-world software. It leads us to collect a set of principles guiding the variability evolution of the Linux kernel and how they ease its maintenance and evolution. We also discuss how our patterns provide concrete scenarios in which existing reasoning techniques yield incorrect results.

9 Future Work (Research Agenda)

Based on our catalog, we formulate the following research directions to extend our current work:

Coevolution Coverage and External Practices As our patterns cover only a small fraction of the whole kernel evolution history, future research shall investigate which other kinds of changes exist in the kernel, which relate to the coevolution of variability models and other artifacts, and how such coevolution occurs. In this direction, further research should investigating coevolution when changes are not triggered by adding or removing features in the variability model (e.g., updating a cross-tree constraint, *ifdef* condition, etc.).

Moreover, as our patterns are an indirect observation of what developers do, it would be valuable to conduct interviews with kernel developers to get further insights on how they coevolve variability models and related artifacts,

and verify whether there are existing practices in the Linux kernel community that corroborate our reported patterns.

Pattern Generality Further research shall verify the generality of our catalog, checking whether our patterns occur in systems other than Linux. This can be achieved by investigating other Kconfig-based variant-rich software systems [9, 44], if scoped to open-source systems, or industrial product lines that have a similar structure as found in the kernel [6].

Evolution Algebra and New Product-Line Theories After defining which patterns are general, a natural follow-up is the decomposition of patterns into a set of operators that transform the variability model, mapping, and code. Patterns, in turn, would be expressed as a mere composition of such operators. The set of derived operators would comprise an evolution algebra for evolving systems whose structure is similar to one found in the Linux kernel. Such algebra could then be supported by specialized tools (e.g., version control systems, IDEs, etc.). Building on top of the evolution algebra, new theories could also be devised, accounting not only feature refinement (as in [10]), but also retirement situations.

Pattern-Based Feature Traceability Our patterns provide a starting point for creating new feature traceability heuristics in systems that follow a similar structure as found in the Linux kernel (variability model, mapping, and C code with annotations). Although existing feature localization techniques [12,15,18, 42,60,66,68] can relate code artifacts (or fragments of them) to features of the system, enabling the vertical traceability between features and code, evolution imposes a *temporal traceability* among features; to trace a feature from a given point to another back in the evolution history or forward in time, one must account for changes that occur together with the variability model; otherwise, incorrect traces might be reported (e.g., as in the case of FB_IMAC, discussed throughout the paper). As we argue in previous work [50], we are unaware of any existing technique that performs such a holistic analysis. In this case, our patterns can serve as a starting point for researching pattern-based traceability heuristics. For example, as reported in our two merge patterns (MVOFS and MVOFNO), the removal of a feature and its implementation artifacts, together with aiding the implementation of another feature with the capabilities of the removed one, is likely to characterize a merge between the two features.

Alternatively, evolution patterns can be incorporated in the evolution process of variant-rich systems. Once cataloged (e.g., following our methodology), patterns can be associated with each new commit, either manually (e.g., by stating such relation in commit log messages), or automatically. In the latter case, research shall investigate how to detect whether patches conform to specific patterns. Associating patterns and commit patches are likely to improve developers' productivity when revisiting a past change and reduce misinterpretations when analyzing its structure.

References

1. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring Product Lines. In: Proceedings of the International Conference on Generative Programming and Component Engineering, pp. 201–210. ACM (2006)
2. Apel, S., Batory, D., Kstner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013)
3. Babar, M.A., Chen, L., Shull, F.: Managing Variability in Software Product Lines. IEEE Software **27**(3), 89–91, 94 (2010)
4. Baresi, L., Guinea, S., Pasquale, L.: Service-Oriented Dynamic Software Product Lines. IEEE Computer **45**(10), 42–48 (2012)
5. Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C., Wise, T.E.: GENESIS: An Extensible Database Management System. IEEE Transactions on Software Engineering **14**(11), 1711–1730 (1988)
6. Berger, T., Nair, D., Rublack, R., Atlee, J.M., Czarnecki, K., Wąsowski, A.: Variability Modeling in Industry: Practices, Benefits, and Challenges. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems. ACM/IEEE (2014). To appear
7. Berger, T., She, S., Lotufo, R., Czarnecki, K., Wąsowski, A.: Feature-to-code Mapping in Two Large Product Lines. Tech. rep., Department of Computer Science, University of Leipzig (2010)
8. Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K.: Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In: Proceedings of the International Conference on Automated Software Engineering, pp. 73–82. ACM (2010)
9. Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K.: A Study of Variability Models and Languages in the Systems Software Domain. IEEE Transactions on Software Engineering **39**(12), 1611–1640 (2013)
10. Borba, P., Teixeira, L., Gheyi, R.: A Theory of Software Product Line Refinement. Theoretical Computer Science **455**(0), 2–30 (2012)
11. Bovet, D., Cesati, M.: Understanding the Linux Kernel. O'Reilly & Associates Inc (2005)
12. Chen, K., Rajlich, V.: Case Study of Feature Location Using Dependence Graph, after 10 Years. In: Proceedings of the International Workshop on Program Comprehension, pp. 1–3. IEEE (2010)
13. Corbet, J., Kroah-Hartman, G., McPherson, A.: Linux Kernel Development: How Fast It is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. `http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2013` (2013). Last seen: July 16th, 2014
14. Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers, 3rd Edition. O'Reilly (2005)
15. Deprez, J.C., Lakhotia, A.: A Formalism to Automate Mapping from Program Features to Code. In: Proceedings of the International Workshop on Program Comprehension, pp. 69–78. IEEE (2000)
16. Dietrich, C., Tartler, R., Schröder-Preikschat, W., Lohmann, D.: A Robust Approach for Variability Extraction from the Linux Build System. In: Proceedings of the International Software Product Line Conference, pp. 21–30. ACM (2012)
17. Dintzner, N., Van Deursen, A., Pinzger, M.: Extracting Feature Model Changes from the Linux Kernel Using FMDiff. In: Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems, pp. 22:1–22:8. ACM (2013)
18. Eisenbarth, T., Koschke, R., Simon, D.: Locating Features in Source Code. IEEE Transactions on Software Engineering **29**(3), 210–224 (2003)
19. Ernst, M.D., Badros, G.J., Notkin, D.: An Empirical Analysis of C Preprocessor Use. IEEE Transactions on Software Engineering **28**(12), 1146–1170 (2002)
20. Favre, J.M.: Understanding in the Large. In: Proceedings of the International Workshop on Program Comprehension, pp. 29–38. IEEE (1997)
21. Feitelson, D.G.: Perpetual Development: a Model of the Linux Kernel Life Cycle. Journal of Systems and Software **85**(4), 859–875 (2012)
22. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)

23. Godfrey, M.W., Tu, Q.: Evolution in Open Source Software: A Case Study. In: Proceedings of the International Conference on Software Maintenance, pp. 131–142. IEEE (2000)
24. Guo, J., Wang, Y., Trinidad, P., Benavides, D.: Consistency Maintenance for Evolving Feature Models. Expert Systems and Applications **39**(5), 4987–4998 (2012)
25. Gurp, J.V., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, pp. 45–54. IEEE (2001)
26. Holdschick, H.: Challenges in the Evolution of Model-Based Software Product Lines in the Automotive Domain. In: Proceedings of the International Workshop on Feature-Oriented Software Development, pp. 70–73. ACM (2012)
27. Izurieta, C., Bieman, J.: The Evolution of FreeBSD and Linux. In: Proceedings of the International Symposium on Empirical Software Engineering, pp. 204–211. ACM (2006)
28. Janota, M., Kiniry, J.: Reasoning About Feature Models in Higher-Order Logic. In: Proceedings of the International Software Product Line Conference, pp. 13–22. IEEE (2007)
29. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
30. Kästner, C., Apel, S.: Virtual Separation of Concerns – A Second Chance for Preprocessors. Journal of Object Technology **8**(6), 59–78 (2009)
31. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: Proceedings of the International Software Product Line Conference, pp. 223–232. IEEE (2007)
32. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: Proceedings of the International Conference on Software Engineering, pp. 311–320. ACM (2008)
33. Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 805–824. ACM (2011)
34. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming, pp. 220–242. Springer (1997)
35. Kim, M., Notkin, D., Grossman, D., Wilson Jr., G.: Identifying and Summarizing Systematic Code Changes via Rule Inference. IEEE Transactions on Software Engineering **39**(1), 45–62 (2013)
36. Kuchana, P.: Software Architecture Design Patterns in Java. Auerbach Publications (2004)
37. Le, D., Walkingshaw, E., Erwig, M.: #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 143–150 (2011)
38. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In: Proceedings of the International Conference on Software Engineering, pp. 105–114. ACM (2010)
39. Liebig, J., Kästner, C., Apel, S.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In: Proceedings of the International Conference on Aspect-oriented Software Development, pp. 191–202. ACM (2011)
40. Lotufo, R., She, S., Berger, T., Czarnecki, K., Wąsowski, A.: Evolution of the Linux Kernel Variability Model. In: Proceedings of the International Conference on Software Product Lines: Going Beyond, pp. 136–150. Springer (2010)
41. Love, R.: Linux Kernel Development, 3rd edn. Addison Wesley (2010)
42. Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergeyev, A.: Static Techniques for Concept Location in Object-Oriented Code. In: Proceedings of the International Workshop on Program Comprehension, pp. 33–42. IEEE (2005)
43. Massa, A.: Embedded Software Development with eCos. Prentice Hall Professional Technical Reference (2002)

44. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Mining Configuration Constraints: Static Analyses and Empirical Results. In: Proceedings of the International Conference on Software Engineering, pp. 140–151. ACM (2014)
45. Nadi, S., Holt, R.: Mining Kbuild to Detect Variability Anomalies in Linux. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 107–116. IEEE (2012)
46. Nadi, S., Holt, R.: The Linux Kernel: A Case Study of Build System Variability. Journal of Software: Evolution and Process (2013)
47. Neill, C.J., Laplante, P.A.: Antipatterns: Identification, Refactoring, and Management. CRC Press (2005)
48. Neves, L., Teixeira, L., Sena, D., Alves, V., Kulezsa, U., Borba, P.: Investigating the Safe Evolution of Software Product Lines. In: Proceedings of the International Conference on Generative Programming and Component Engineering, pp. 33–42. ACM (2011)
49. Passos, L., Czarnecki, K.: A Dataset of Feature Additions and Feature Removals from the Linux Kernel. In: Proceedings of the Working Conference on Mining Software Repositories, pp. 376–379. ACM (2014)
50. Passos, L., Czarnecki, K., Apel, S., Wąsowski, A., Kästner, C., Guo, J.: Feature-Oriented Software Evolution. In: Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems, pp. 17:1–17:8. ACM (2013)
51. Passos, L., Czarnecki, K., Wąsowski, A.: Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In: Proceedings of the International Workshop on Feature-Oriented Software Development, pp. 62–69. ACM (2012)
52. Passos, L., Guo, J., Teixeira, L., Czarnecki, K., Wąsowski, A., Borba, P.: Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In: Proceedings of the International Software Product Line Conference, pp. 91–100. ACM (2013)
53. Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S.: Model-driven Support for Product Line Evolution on Feature Level. Journal of Systems and Software **85**(10), 2261–2274 (2012)
54. Rosenmuller, M., Apel, S., Leich, T., Saake, G.: Tailor-made Data Management for Embedded Systems: A Case Study on Berkeley DB. Data & Knowledge Engineering **68**(12), 1493–1512 (2009)
55. Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O., Saake, G.: FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In: Proceedings of the EDBT Workshop on Software Engineering for Tailor-made Data Management, pp. 1–6. ACM (2008)
56. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples, 1st edn. Wiley Publishing (2012)
57. Seidl, C., Heidenreich, F., Aßmann, U.: Coevolution of Models and Feature Mapping in Software Product Lines. In: Proceedings of the International Software Product Line Conference, pp. 76–85. ACM (2012)
58. She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K.: The Variability Model of the Linux Kernel. In: Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems, pp. 45–51. Universität Duisburg-Essen (2010)
59. She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K.: Reverse Engineering Feature Models. In: Proceedings of the International Conference on Software Engineering, pp. 461–470. ACM (2011)
60. Simmons, S., Edwards, D., Wilde, N., Homan, J., Groble, M.: Industrial Tools for the Feature Location Problem: an Exploratory Study. Journal of Software Maintenance and Evolution **18**(6), 457–474 (2006)
61. Spencer, H., Collyer, G.: #ifdef Considered Harmful, or Portability Experience with C News. In: Proceedings of the USENIX Security Symposium (1992)
62. Sun, J., Zhang, H., Wang, H.: Formal Semantics and Verification for Feature Modeling. In: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, pp. 303–312. IEEE (2005)
63. Tartler, R., Sincero, J., Dietrich, C., Schröder-Preikschat, W., Lohmann, D.: Revealing and Repairing Configuration Inconsistencies in Large Scale System Software. International Journal on Software Tools for Technology Transfer **14**(5), 531–551 (2012)

64. Thüm, T., Batory, D., Kastner, C.: Reasoning About Edits to Feature Models. In: Proceedings of the International Conference on Software Engineering, pp. 254–264. IEEE (2009)
65. Thüm, T., Kastner, C., Erdweg, S., Siegmund, N.: Abstract Features in Feature Modeling. In: Proceedings of the International Software Product Line Conference, pp. 191–200. IEEE (2011)
66. Valente, M.T., Borges, V., Passos, L.: A Semi-Automatic Approach for Extracting Software Product Lines. IEEE Transactions on Software Engineering **38**(4), 737–754 (2012)
67. Venkateswaran, S.: Essential Linux Device Drivers, 1st edn. Prentice Hall Press (2008)
68. Wilde, N., Scully, M.C.: Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance **7**(1), 49–62 (1995)
69. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux Kernel. In: Proceedings of the USENIX Security Symposium, pp. 17–31. USENIX Association (2002)