Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel

Leonardo Passos*
University of Waterloo
lpassos@gsd.uwaterloo.ca

Jianmei Guo
University of Waterloo
gjm@gsd.uwaterloo.ca

Leopoldo Teixeira†
Federal University of Pernambuco
lmt@cin.ufpe.br

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

Andrzej Wąsowski‡
IT University of Copenhagen
wasowski@itu.dk

Paulo Borba
Federal University of Pernambuco
phmb@cin.ufpe.br

ABSTRACT

Variability-aware systems are subject to the coevolution of variability models and related artifacts. Surprisingly, little knowledge exists to understand such coevolution in practice. This shortage is directly reflected in existing approaches and tools for variability management, as they fail to provide effective support for such a coevolution. To understand how variability models and related artifacts coevolve in a large and complex real-world variability-aware system, we inspect over 500 Linux kernel commits spanning almost four years of development. We collect a catalog of evolution patterns, capturing the coevolution of the Linux kernel variability model, Makefiles, and C source code. Further, we extract general findings to guide further research and tool development.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.13 [**Reusable Software**]: Domain engineering

General Terms

Design, Documentation, Management

Keywords

Variability, Evolution, Catalog, Patterns, Linux

*Funded by CAPES, grant BEX 0459-10-0.

†Partially supported by the National Institute of Science and Technology for Software Engineering (INES).

‡Supported by Danish Council for Independent research under VARIETE project.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPLC '2013 Tokyo, Japan

Copyright 2013 ACM 978-1-4503-1968-3/13/08 ...$15.00

http://dx.doi.org/10.1145/2491627.2491628.

1. INTRODUCTION

Many real-world software systems, such as the Linux kernel, the embedded operating system eCos, and various software product lines (SPLs),[1] leverage variability modeling to achieve systematical reuse and mass customization [21]. These variability-aware systems often specify configurable system options relevant to users as *feature declarations*, or features for short, and provide *variability models* to manage all features and their relationships. Such features may then be referred in related artifacts (e.g., Makefiles and C source code) by means of explicit *variation points* (e.g., ifdefs).

Large variability-aware systems contain complicated variabilities. Along with the complexity of their variability models [23, 19], these systems have hundreds of variation points spread across many files. For instance, the Linux kernel (release 3.3) contains over 12,000 features and above 80,000 compile-time variation points, distributed in more than 1,600 Makefiles and 30,000 C implementation and header files. As Nadi and Holt show [16], only 16% of the features in the Linux kernel are exclusive to the variability model; 49% are used in Makefiles, 17% control variability in source code, and 18% are used in compile-time variation points in both Makefiles and source code.

In such settings, the coevolution of variability models and different artifacts is inevitable. The interrelation of different sources of variability makes variability evolution intricate. A thorough analysis of a specific Linux kernel release 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 [18].

Until now, little is known about the coevolution of variability models and related artifacts [22], a pre-requisite for providing better tools and practices. Existing research mostly focuses on techniques and case studies regarding the evolution of variability models alone [1, 26, 20, 10, 15]. These techniques are evaluated in terms of fictitious evolution scenarios or randomly generated models. Unlike these, Neves et al. [17] perform a comprehensive study of the coevolution of variability models and related artifacts in real SPLs. Their investigation, however, is restricted to small systems (at most 40 features), which may not be representative of the com-

[1] http://tinyurl.com/sei-casestudies

plexity typically found in large variability-aware systems [25, 2]. Moreover, the authors only cover operations that conform to the theory of SPL refinement [4], i.e., operations that preserve the behavior of existing products and guarantee that every product prior to such operations can still be mapped to a corresponding product in the resulting SPL. Such situation, however, is too restrictive in practical settings, where feature retirement often arises [18].

To better understand how variability models and related artifacts coevolve, we study the evolution history of a large and complex real-world variability-aware system: the Linux kernel. We inspect over 500 commits relative to the addition and removal of features, spanning almost four years of Linux kernel development. We collect a catalog of high-level *evolution patterns* capturing the coevolution of the Linux kernel variability model, Makefiles, and C source code. For each pattern, we crosscheck specific properties of its instances against evidence from existing literature, and report a set of findings to guide further research and tool development for the coevolution of variability models and related artifacts in variability-aware systems. We claim the extracted catalog and the reported findings as the two main contributions of this paper. We hope that these results will help improve future methods and tools for engineering variability-aware systems.

2. BACKGROUND

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

2.1 The Three Spaces of the Linux Kernel

Variability in the Linux kernel is present in three spaces: *variability model*, *mapping*, and *implementation*. Following the steps in Fig. 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.[2] 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).

As shown in the file fragment from the variability model in Fig. 1, features in Kconfig are represented mostly by configs (lines vm3 and vm5). In our example, FB (the parent of all frame-buffer-related features)[3] 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), and the configurator uses it 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 enclosing 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 enclosing feature. A select attribute is a reverse dependency that enforces immediate selection of 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).

Once users finish their selection, they save their configuration. The configurator then writes a .config file (step 3), containing a sequence of feature-name=value lines. Note that 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, controls the whole compilation process of the kernel. To build a kernel image realizing a given configuration, users invoke make (step 4), which triggers the execution of the top Makefile in 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 make 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 other elements can be appended to. When appending directories (line 3 above), Kbuild recursively runs the Makefile in each such directory and generates all objects of a special list: obj-y (similarly, a list obj-m is kept for dynamic loadable modules). Objects are conditionally added to such a list by replacing y with a feature name. As shown in the Makefile fragment of Fig. 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 matching the same name. However, that is not always the case. For fb.o, there is no fb.c file in the Makefile's directory, so Kbuild relies on the presence of 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).

Implementation. Variability in the source code base is expressed in terms of conditional compilation macro directives (ifdefs), whose conditions are essentially Boolean expressions over feature names. These directives guard whether certain source code fragments should be compiled.

Prior to compilation, Kbuild assures that the C compiler (cc) appends an inclusion directive of autoconf.h in the corresponding target source file (step 5.3) prior to calling cpp. This header contains macro definitions for the features selected during configuration and 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 Boolean/tristate features are set to 1; integer/string features, if present, lead to macros whose values match those given at configuration.

Given the macro definitions in autoconf.h, cpp then evaluates all code guards directives, effectively deciding code blocks to

[2] https://www.kernel.org/doc/Documentation/kbuild/
[3] http://tinyurl.com/framebuffer-txt

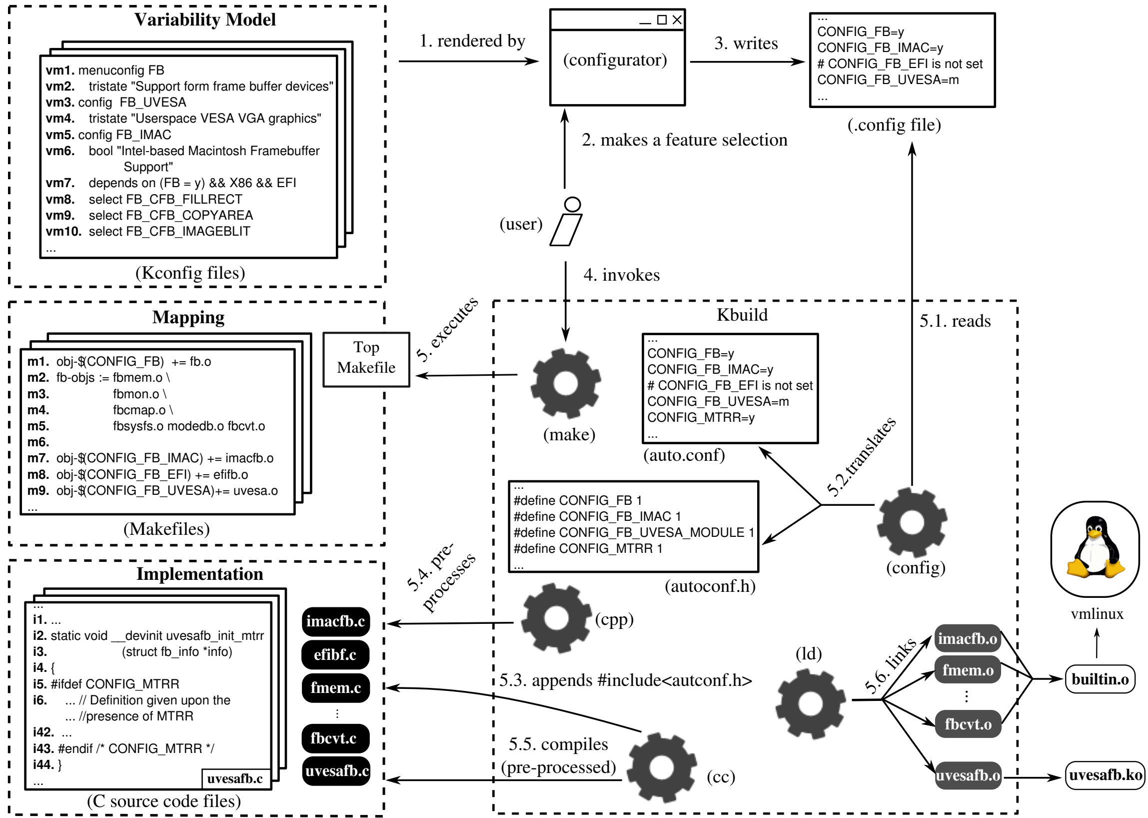

Figure 1: The three spaces in the Linux kernel and their interaction with Kbuild

be included and those to be removed (step 5.4). The C compiler, in turn, compiles the resulting code (step 5.5). From the example configuration in Fig. 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 gathers changes in each space and shows how they coevolve as a result.

As a walk-through to our notation, consider a particular feature merge pattern (Fig. 2). An instance of the pattern is the case where developers add the capabilities of FB_IMAC into FB_EFI, and consequently remove FB_IMAC from the variability model, mapping and implementation.[4]

As shown in the figure, a pattern depicts the transition from a *before-state* to a state after the application of the prescribed change—*after-state*. The transition is denoted 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). In the before-state of Fig. 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:

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

In a build rule (e, r_1, r_2), e is an expression E over feature names; 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. To avoid clutter, the condition and the alternative rule may be omitted to represent unconditional rules (e is taken as true and r_2 is an empty rule—ϵ). Moreover, the shorthand form (e, r_1) is used when r_2 is empty. The pattern in Fig. 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

[4] Commit 7c08c9ae.

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 "..." to ignore unrelated elements that do not affect the features under analysis.

In the after-state of the merge pattern in Fig. 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 $f_2 > f_1$.

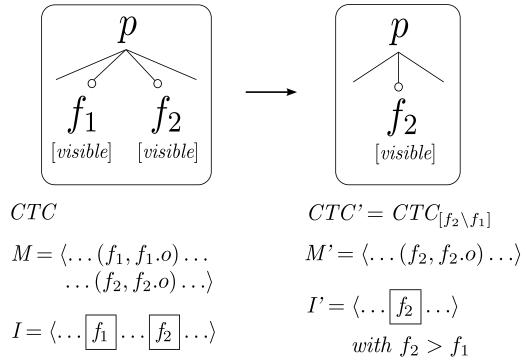

Figure 2: Merge Visible Optional Feature into Sibling

3. METHODOLOGY

We extract our pattern catalog by analyzing patches 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, as its variability model follows the same growth pattern of the variability model of the whole kernel [15].

We collect the entire set of added and removed features by calculating the feature set difference among pairs of consecutive stable kernel releases. To that end, we extract the Kconfig infrastructure shipped in the Linux kernel source code to list the features in each kernel release. Currently, such 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 conducted our analysis.

Next, we create two random samples: one comprising 5% (206) of all added features, and another with 10% (101) of all removed features. An entry in each sample contains the feature name and the kernel release that either adds or removes such a feature. 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). Although differing in terms of the underlying release population, these sizes are consistent with [15], showing that feature additions in the Linux kernel exceed feature removals.

By parsing all the patches in the version control system (VCS) of the Linux kernel, we link each sample entry with the commit, referred as *primary commit*, that adds or removes

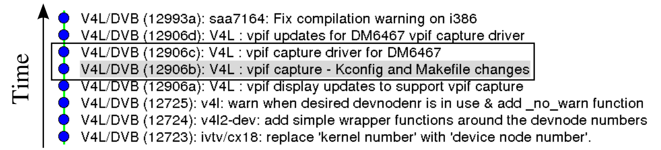

Figure 3: Commit window example

the corresponding feature (named as *primary feature*). As one primary feature associates with a single primary commit, we collect 206 and 101 primary commits relative to added and removed features, respectively.

Once all primary commits are known, we set to extract the evolution patterns. We describe their extraction procedure in the following.

Pattern Extraction

We apply a multi-step analysis to retrieve the evolution pattern of a primary feature.

As the primary commit only guarantees to retrieve changes in the configuration space (changes in other spaces may be in other commits), we rely on a *commit window* to collect commits related to the primary feature that either follow or precede the primary commit. To illustrate this, consider the addition of the primary feature CAPTURE_DAVINCI_DM64X. Figure 3 shows a sequence of commits (one commit per line) changing the V4L and DVB subsystems,[5] as recorded in the Linux kernel VCS. In the figure, the primary commit is highlighted in gray, with its patch shown in Fig. 4. 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 adds a Kconfig entry (Fig. 4, lines 8–11) and a new compilation rule in the correct Makefile (lines 15–16). As seen in the patch, the developer does not add a file vpif_capture.c. 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 it is included in the commit window, shown as a black rectangle in Fig. 3.

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 these problems, we restrict the maximum size of a commit window to contain at most 40 commits, and starting from the primary commit, we expand a commit window using four main rules: (i) include commits that add/remove compilation units known to be mapped to the primary feature; (ii) include commits whose changes affect files mapped to the primary feature; (iii) include commits whose changes add/remove compile-time variation points that rely on the primary feature; (iv) include commits that modify the declaration of the primary feature in the variability model. Restricting the window size works well for our samples, as on average it is 1.65.

Within the 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. This classification may ignore changes unrelated to the primary feature (e.g., lines 5–6 in Fig. 4). Windows with the same category are then clustered together.

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

```
1  drivers/media/video/Kconfig
2
3  config DISPLAY_DAVINCI_DM646X_EVM
4     help
5  -     Support for DaVinci based display device.
6  +     Support for DM6467 based display device.
7
8  +config CAPTURE_DAVINCI_DM646X_EVM
9  +   tristate "DM646x EVM Video Capture"
10 +   depends on VIDEO_DEV && MACH_DAVINCI_DM6467_EVM
11 +   ...
12
13 drivers/media/video/davinci/Makefile
14
15 +obj-$(CONFIG_CAPTURE_DAVINCI_DM646X_EVM) +=  \
16     vpif_capture.o
```

Figure 4: Patch adding the Davinci D6467 driver (primary commit)

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 or not; (b) type: whether the feature is a *switch* (i.e., Boolean/tristate) or a *value-based* feature (int/string) [3]; (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. We then re-cluster results accordingly and discard clusters with less than 3% of the size of the sample under analysis. We choose this cut as it matches exactly three instances in the removal sample, thus conforming to the *Rule of Three*: *a pattern can be called a pattern only if it has been applied to a real world solution at least three times.*[6] Applying such a percentage in the addition sample results in finding at least six instances, as that sample contains roughly twice more primary commits than the removals sample.

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 508 commits in all commit windows, where 406 relate to features in the additions sample and the remaining 102 to features in the removals sample. It is worth noting that some commit windows do not lead to sound conclusions, and are thus excluded from analysis: these windows account for 3% (6) of the features in additions sample, and 7% (7) of the features in the removals sample.

Two authors were responsible for collecting patterns. As this process requires human analysis (e.g., establishing the boundaries of a commit window and the subcategories used for clustering), each of those authors reviewed the results of the other; in the case of inconsistencies, the authors discussed them and reached a consensus on the correct form of the pattern. Later, a third author reviewed all results, pointing out possible inaccuracies. In that case, all three authors discussed any resulting issue and reached a final agreement on the patterns herein reported.

All the collected data, its analyses and the custom underlying infrastructure are available at our website.[7]

[6] http://c2.com/cgi/wiki?RuleOfThree

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

Additions sample		Removals sample	
Pattern	**# of instances**	**Pattern**	**# of instances**
AVOMF	95	RVOMF	22
AVOGMF	10	RVOGMF	9
AVONMF	26	RVONMF	9
AIMF	13	RIMF	3
FTC	8	MVOFNO	3
		MVOFCI	3
		MVOFS	3
RNM	9	RNM	17
Total	161	**Total**	69
Sample %	78	**Sample %**	68

Table 1: Patterns frequency

4. PATTERN CATALOG

This section presents 12 patterns in our catalog. Table 1 shows their usage frequency in each associated sample. We discuss all the patterns in the following, except for rename (RNM), which we omit due to spacing.

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.

As shown in Fig. 5, the pattern adds a new optional and visible feature f in the variability model, along with its associated constraints. 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 Sect. 3, is an instance of this pattern.

Most primary features in the additions sample (46%) fit into this pattern and are used to add: (a) *device drivers* (90%)—features that exist inside the kernel space and are "plugged-in" to the core kernel (subsystems + system calls) to support different hardware. Such high frequency agrees with existing work [9, 12, 15, 8], stating that the Linux kernel evolution is mainly driven by the addition of new drivers; (b) *core functionality* (2%)—features added to the core kernel, such as supporting self-test for 64-bit atomic instructions; (c) *complementary functionality* (5%)—features that complement an existing feature, either in or outside the core, with extra functionality (e.g., debugging support); and (d) *shared functionality* (3%)—features that provide a common base to a set of features. For example, the CAN (Controller Area Network) network bus driver provides a generic infrastructure that all specific CAN drivers rely on.

Instances of this pattern are either tristate (90%) or Boolean. The dominance of tristate features follows a trend in most of the patterns related to modular features, evidencing a tight relation between the two. That relation is intentional, as modular tristate features provide flexibility to cover different requirements and configuration purposes (e.g., in embedded platforms where hardware can be anticipated, tristate features can be statically linked against the final kernel; in other situations, when hardware changes at will, tristate features can be compiled as modules and loaded as needed).

It is worth noting that modular features can still be referenced in code extensions (ifdefs) elsewhere. To verify the existence of such extensions while overcoming the scope limitation imposed by the commit window size, we collect all

ifdefs in the kernel releases that adds visible optional modular features found in our sample. We find that such extensions are infrequent, as they only appear in five device drivers among all added visible modular features, and two extensions appear in a header file exporting certain functions of a new driver. In that case, an ifdef checks whether the driver is present. If not, an alternative code generates a warning stating that calling the exported function leads to an error; the remaining three appear outside the related driver code, but still outside the core kernel. This evidences that adding new drivers agrees with the kernel planned architecture, as drivers should integrate to the core by registering themselves as handlers to specific events (e.g., hardware interrupts) [5].

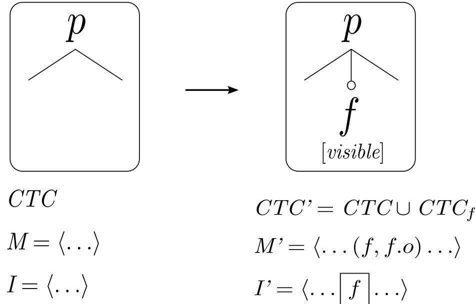

Figure 5: Add Visible Optional Modular Feature

Add Visible Optional Guard Modular Feature (AVOGMF). This pattern is a variant of AVOMF. The only difference with the former is that in this pattern f serves as a compilation guard over an entire directory, controlling whether the compilation process should recursively descend to it. As such, it contains an additional mapping rule

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

instructing Kbuild to enter a child directory $f/$ upon the presence of that feature. There Kbuild processes a Makefile with the rule on how to build f itself. The addition of the device driver supporting Realtek's© 8192 network adapter illustrates this (see Fig. 6): in the parent Makefile (top snippet in the figure), Kbuild assesses whether RTL8192 is present. If so, it enters the rtl8192se directory and processes the child Makefile in there (bottom snippet); in that case, RTL8192's presence enables the compilation of all objects in the rtl8192se-objs list.

This pattern comprises 5% 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 70% of the instances of in this pattern; (b) in the remaining, a guard modular feature roots a subtree with at least one modular descendant feature. All modular features in the subtree reside in the directory $f/$.

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; 13% of the additions instances fit this pattern.

As shown in Fig. 7, this pattern adds a visible optional

```
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
...
```

Figure 6: Guarded directory example (AVOGMF)

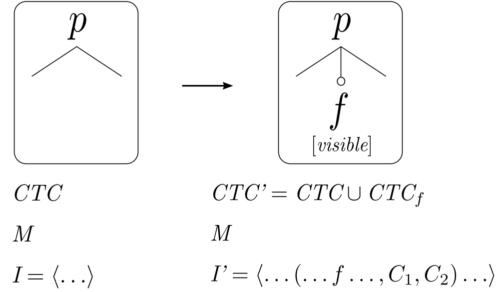

Figure 7: Add Visible Optional Non-Modular Feature

feature in the feature model, while not changing the mapping. The implementation changes by now 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:

```
+#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[8] is set to four kilobytes; otherwise it is set to one kilobyte.

Following the granularity measures proposed by existing studies [14], from a total of 108 code extensions (ifdefs) in the commit windows of all non-modular features in this pattern, 44.4% are extensions at the global level (e.g., declaring a new macro, variable, function, structure, etc.), 33.3% occur at the function level (e.g., by adding statements inside a function), 14% extend a block statement (e.g., adding a statement inside an if-block) and 8.3% extend a type declaration (e.g., adding a field to a structure). This distribution is similar to the one found by Liebig et al. [14] when investigating 40 preprocessor-based systems. As we do not find any extensions on the level of single statements (e.g., changing the type of a variable declaration), expressions or function signatures (e.g., adding a conditional parameter), our findings strengthens their claim that fine-grained extensions are not frequent in practice. Interestingly, f negatively affects the conditionally compiled code in 2% 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).

Opposed to modular features, in 92% of the instances

[8] http://squashfs.sourceforge.net/

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 it dynamically at runtime. The only situation in which f is tristate is when it contains a reverse dependency towards a modular tristate feature f_s; if put 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.

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.

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

As AVOMF, this pattern covers the addition of new functionality, and comprises 6% of all additions.

Featurize Code (FTC). Featurization results from creating a feature from existing elements, and covers 4% of features in the additions sample.

In the extracted pattern, illustrated in Fig. 8, the compilation of a set of object files $f_1.o \ldots f_n.o$ depends upon the presence of a feature p. This pattern is applied when a given object file $f_i.o$ ($1 \leq i \leq n$) is not essential to the functionality provided by p; rather, its capability is optional. In that 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, becomes a child of p in the hierarchy, and $f_i.o$ is removed from the list of objects controlled by p. Adding f_i gives users a finer-grained control over the configuration process, while decreasing the granularity of p. That prevents unnecessary features to be shipped in the resulting kernel, and in turn, improves its memory usage and boot time.

The example shown in Fig. 9 illustrates the featurization of me4000.o, previously controlled by COMEDI_PCI_DRIVERS, into the new feature COMEDI_ME4000.

Retire Feature (RVOMF, RVOGMF, RVONMF, RIMF). Retiring a feature consists in removing it from all the spaces it appears. Four retirements patterns occur in the removals sample: (a) *Retire Visible Optional Modular Feature (RVOMF)*; (b) *Retire Visible Optional Guard Modular Feature (RVOGMF)*; (c) *Retire Visible Optional Non-Modular Feature (RVONMF)*; (d) *Retire Internal Modular Feature (RIMF)*. These patterns are the inverse of their counterpart addition patterns (namely AVOMF, AVOGMF, AVONMF, and AIMF) with a similar frequency distribution. Due to spacing, we omit their representation.

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

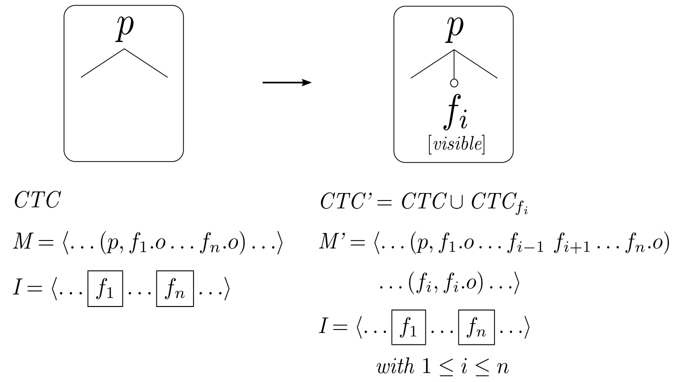

Figure 8: Featurize Code

```
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
```

Figure 9: Featurize Code example

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) a feature supersedes an obsolete one, which is then retired.

Interestingly, 67% of RVONMF and RIMF, and 23% of the RVOMF instances are removed as a consequence of retiring the whole subtree containing them. This suggests that: (i) most visible optional non-modular and internal modular features are retired together with their parent; (ii) some forms of retirement occur in a coarse-grained manner and are triggered by the removal of a feature rooting an entire subtree. Thus, its descendants are recursively removed.

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 10 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.

Of all instances in the removals sample, 3% fit into this pattern and mostly 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. As shown in the related patch (see Fig. 11), developers drop the original cross-tree constraints and rename the previous feature from the variability model

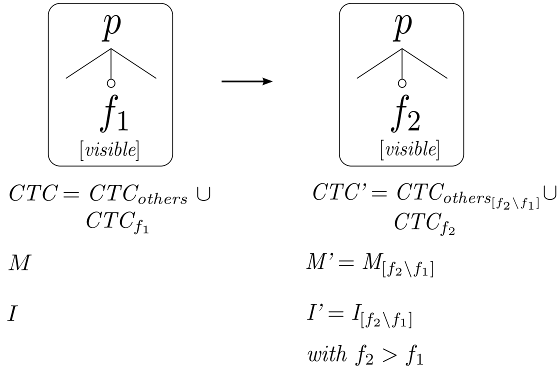

Figure 10: Merge Visible Optional Feature into New One

```
a/drivers/power/Kconfig b/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
...
```

Figure 11: Merge Visible Optional Feature into New One example

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. That follows from the fact that merges can cause the false impression that some features cease to exist.

Merge Visible Optional Feature into Computed Internal (MVOFCI). This pattern, depicted in Fig. 12, removes the user decision about the presence of a visible optional feature f_1, whose selection triggers an associated internal feature f_2. In the after-state, the presence of f_2 becomes computed (shown with an annotation C) by the conjunction of all constraints imposed by f_1 and f_2. Feature f_1 is then removed from the variability model, and all references to f_1 are updated to f_2. In total, this pattern accounts for 3% of all removals.

The patch snippet in Fig. 13 illustrates the pattern: RT2400PCI_LEDS (matches f_1) is removed from the variability model, while RT2X00_LIB_LEDS (matches f_2), previously selected by RT2400PCI_LEDS, becomes a computed feature. Its presence is now given by the default value y (users cannot configure the feature, as it is invisible) iff RT2X00_LIB, NEW_LEDS, and LEDS_CLASS are all present.

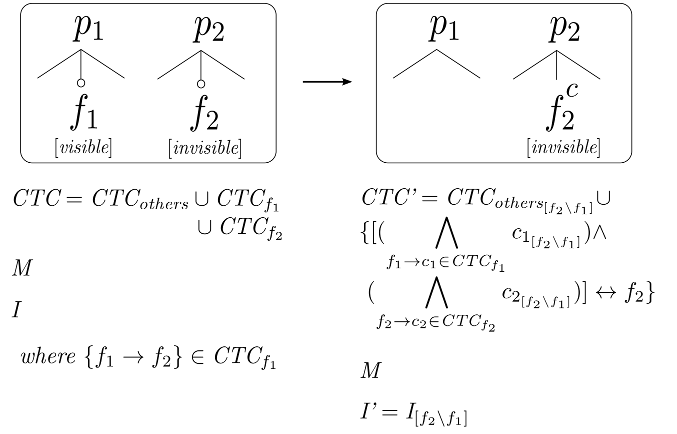

Figure 12: Merge Visible Optional Feature into Computed Internal

Merge Visible Optional Feature into Sibling (MVOFS). This pattern covers the situation in which developers merge a visible optional feature into its sibling (see Fig. 2), due to their similarity (the merging of the features FB_IMAC and FB_EFI, previously discussed, exemplifies the pattern).

This pattern aims at easing maintenance, as keeping the two similar features potentially requires a duplicate effort whenever a change occurs in either of them. As other merges, this pattern responds for 3% of all removals.

Others. The patterns reported in Tab. 1 cover most of the additions (78%) and removals (68%) of our sample.

The remaining commits that were not excluded from analysis in the additions sample refer to two situations: (i) clusters under the threshold: these include features that exist with the sole purpose of adding a compilation flag (2%), that guard the compilation of a given directory (2%), that exist only in the variability model (2%), that are non-optional modular features (1%), or that are merges (1%); (ii) clusters above the threshold, but that do not fit in a pattern: these include non-modular features (4%) that are invisible or are value-based, or refer to featurizations (7%).

For the most part, unexcluded instances in the removals sample refer to distinct situations in which features are removed from the variability model, but remain mandatory in code (7%). Another cluster (3%) actually fits in a well-defined pattern (*Retire Computed Internal Non-Modular Feature*), but we refrained from including it on the basis that an inverse pattern in the additions sample does not exist (no feature can exist without being first added). The remaining instances refer to nine small clusters (1% each), or are the residuum of clusters contributing to patterns, but that do not share all the characteristics of such patterns (6%). As in the addition sample, few features are value-based (6%), or are either mandatory or computed (11%).

5. FINDINGS

We summarize our findings as follows. (i) Most features in the Linux kernel are modular and cause little scattering when added. Hence, we conjecture that the practice of keeping high modularity and low scattering mitigates the negative effects of ifdef-based variation points [24, 7, 6], allowing the kernel to continuously evolve. Furthermore, as the AVONMF pattern shows (the second highest pattern found)

```
drivers/net/wireless/rt2x00/Kconfig

-config RT2400PCI_LEDS
-  bool "Ralink rt2400 leds support"
-  depends on RT2400PCI && NEW_LEDS
-  select LEDS_CLASS, RT2X00_LIB_LEDS
-  ---help---
-   This adds support for led triggers provided my mac80211.

config RT2X00_LIB_LEDS
    boolean
-   depends on RT2X00_LIB && NEW_LEDS
+   depends on RT2X00_LIB && NEW_LEDS && LEDS_CLASS
+   default y
...

drivers/net/wireless/rt2x00/rt2400pci.c

...
-#ifdef CONFIG_RT2400PCI_LEDS
+#ifdef CONFIG_RT2X00_LIB_LEDS
 value = rt2x00_get_field16(eeprom, EEPROM_ANTENNA_LED_MODE);
 rt2400pci_init_led(rt2x00dev, &rt2x00dev->led_radio,
                   LED_TYPE_RADIO);
 ...
-#endif /* CONFIG_RT2400PCI_LEDS */
+#endif /* CONFIG_RT2X00_LIB_LEDS */
...
```

Figure 13: Merge Visible Optional Feature into Computed Internal example

adding non-modular features does not cause fine-grained extensions to appear (e.g., controlling whether a function parameter should or not be declared), which evidences a disciplined annotation usage. In turn, it greatly mitigates the drawbacks of the low-level variability representation used in the kernel. (ii) Despite the size and complexity of the Linux kernel, adding and removing features is performed in a systematic manner, leading to a small number of patterns. This evidences that variability evolution can follow systematic patterns. Following Kim et al. [13], patterns could then be retrieved from the evolution history (as we did) and used to summarize changes. (iii) Our merge patterns show that considering changes only in the variability model may lead to incorrect conclusions. While removed from the variability model, these features are combined with existing features by means of changes in other spaces, with no overall functionality loss. Consequently, existing traceability and edit-reasoning techniques [26] that account for changes only in the variability model have to be adapted to consider coevolution with other artifacts. (iv) Although we do not prescribe any specific support mechanism, our catalog evidences to tool implementers which operations developers face in practice when coevolving variability models, build files and source code. Furthermore, our retirement patterns evidence that developers remove entire subtrees, along with their related artifacts. Therefore, existing tools ought to support such functionality. (v) We provide further evidence towards the claim in [18] stating the need to extend the existing theory of SPL refinement [4], as feature retirement is rather frequent.

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, we rely on randomly collected samples and on a threshold cut to eliminate non-recurrent change strategies.

Scoping to x86 architecture is also a threat. Yet, previous research [15] reports that the variability model of x86 has a similar growth to the variability model of the whole kernel.

The choice of the threshold value also poses a threat. We argue, however, that the use of 3% assures the inclusion of less frequent patterns, while still requiring a minimal recurrence. Furthermore, the threshold prevents us from incorrectly reporting patterns over extreme outliers.

The size of the samples also poses a threat, although minor: if a different and larger sample is used, new patterns may be found, possibly with a different relative frequency. However, our patterns would still be valid (although not possibly reported), as they follow from the threshold value.

Manually extracting and classifying patterns also poses a threat. To mitigate this, we devise a methodology with a well-defined sequence of steps. As some of them rely on certain subjectivity, we perform three extensive reviews (see Sect. 3) to guarantee consistency among all patterns. Moreover, all our collected data and analyses are publicly available, and we encourage the community to verify it independently.

As an external threat, we cannot state that our patterns occur in systems other than the Linux kernel, even if based on Kconfig+Kbuild. We argue, however, that this does not diminish the importance of our catalog; Linux is the most successful open-source software, with a large and rich variability whose evolution can provide new insights on how to support evolution in a multi-space setting.

7. RELATED WORK

Our previous investigation [18] presents four preliminary evolution patterns taken from a specifically-selected set of 140 commits. In contrast, this work bases on a random set of over 500 commits and collects 13 evolution patterns and their usage frequency following a systematical methodology. Further, we crosscheck them against existing research and extract general findings.

She et al. [23] 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. [15] 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. The authors, however, restrict their analysis to variability models, which, as we argued before, leads to an incomplete and possibly misleading understanding of the evolution in place.

Others cover evolution in a multi-space setting, but restrict analysis to small SPLs. Holdschick [11] presents change operations between variability models and functional models in the automotive domain. Neves et al. [17] extract operations conforming to the refinement theory in [4]. Their operations guarantee that old products can still be mapped to products in the SPL 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.

Researchers also investigate the problems resulting from the coevolution of the spaces of the Linux kernel. Tartler et

al. [25] 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. [16] 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).

Seidl et al. [22] present a set of evolution scenarios and mapping operators to reestablish the correct binding of different spaces in an SPL. Opposed to our work, the authors do not provide 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. As we show, that is not the case in practice, as code is featurized (see the FTC pattern).

Kim et al. [13] 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 AND FUTURE WORK

We investigate the coevolution of the Linux kernel variability model, Makefiles and C source code by inspecting a sample of the evolution history spanning almost four years of Linux kernel development. We collect a catalog of evolution patterns that capture a range of high-level evolution operations as a result of the coevolution process.

Each pattern in our catalog covers how changes affect each artifact, the usage frequency of each pattern, and how they are used by Linux kernel developers. To the best of our knowledge, this work is the first to study the coevolution of variability models and related artifacts in a multi-space setting of a large and complex real-world software. Further, we extract a set of general findings to guide further research and tool development for variability coevolution.

As future work, we aim to investigate 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.). Furthermore, we aim to extract a set of general operators capable of expressing patterns in any ifdef-based project.

9. REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE*, 2006.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the systems software domain. Technical report, Generative Software Development Laboratory, University of Waterloo, 2012.

[3] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *ASE*, 2010.

[4] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *ICTAC*, 2010.

[5] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly, 2005.

[6] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12), 2002.

[7] J.-M. Favre. Understanding-in-the-large. In *IWPC*, 1997.

[8] D. G. Feitelson. Perpetual development: a model of the Linux kernel life cycle. *J. Syst. Softw.*, 85(4), 2012.

[9] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM*, 2000.

[10] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. Consistency maintenance for evolving feature models. *Expert Syst. Appl.*, 39(5), 2012.

[11] H. Holdschick. Challenges in the evolution of model-based software product lines in the automotive domain. In *FOSD*, 2012.

[12] C. Izurieta and J. Bieman. The evolution of FreeBSD and Linux. In *ISESE*, 2006.

[13] M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Trans. Softw. Eng.*, 39(1), 2013.

[14] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE*, 2010.

[15] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux kernel variability model. In *SPLC*, 2010.

[16] S. Nadi and R. Holt. The Linux kernel: a case study of build system variability. *J. Softw. Maint. Evol.-R*, To appear.

[17] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the safe evolution of software product lines. In *GPCE*, 2011.

[18] L. Passos, K. Czarnecki, and A. Wasowski. Towards a catalog of variability evolution patterns: the Linux kernel case. In *FOSD*, 2012.

[19] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wąsowski. A study of non-boolean constraints in variability models of an embedded operating system. In *SPLC*, 2011.

[20] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven support for product line evolution on feature level. *J. Syst. Softw.*, 85(10), 2012.

[21] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

[22] C. Seidl, F. Heidenreich, and U. Assmann. Co-evolution of models and feature mapping in software product lines. In *SPLC*, 2012.

[23] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the Linux kernel. In *VaMoS*, 2010.

[24] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *USENIX*, 1992.

[25] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *Int. J. Softw. Tools Technol. Transf.*, 2012.

[26] T. Thum, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *ICSE*, 2009.