Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case

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

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

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

ABSTRACT

A complete understanding of evolution of variability requires analysis over all project spaces that contain it: source code, build system and the variability model. Aiming at better understanding of how complex variant-rich software evolve, we set to study one, the Linux kernel, in detail. We qualitatively analyze a number of evolution steps in the kernel history and present our findings as a preliminary sample of a catalog of evolution patterns. Our patterns focus on how the variability evolves when features are removed from the variability model, but are kept as part of the software. The identified patterns relate changes to the variability model, the build system, and implementation code. Despite preliminary, they already indicate evolution steps that have not been captured by prior studies, both empirical and theoretical.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

General Terms

Design

Keywords

variability, patterns, evolution, software product lines, Linux

1. INTRODUCTION

Variability evolution is a core point in evolving software product lines [6]. Changes in the variability dictate which features are obsolete, which are new, which products are still possible to be generated, which dependencies still hold, etc. Despite its importance, the Software Product Line community has little knowledge on how variability evolution occurs in practice and which changes are performed when realizing them. The few existing studies do not take feature removal into account [4, 5, 12], while others [14, 8] focus on the variability model alone. Altogether, they fail to cover the variability evolution when features are removed from the variability model, while still being kept part of the software. To address this issue, we study a real world variant rich software –the Linux kernel– and extract evolution patterns describing how variability evolves across different artifacts (variability model, build files, and source code) when features are erased from the variability model, but not from the software itself.

The Linux kernel is the most successful open source software, containing a rich and extensive variability that allows it to support a large range of architectures, device drivers and application domains [15].

Variability in the Linux kernel is vertically present in three separate, but related spaces [10]: *configuration space*: kernel configuration files (Kconfig), comprising the Linux variability model; *compilation space*: kernel build files (KBuild), mostly written as Makefiles with implicit rules [16]; *implementation space*: realization of all features, mostly written as C code.

The Linux kernel configuration space was first studied by She et al. [14], who analyze and compare its complexity with regards to existing models in SPLOT [9]. Lotufo et al. [8] extend that work by a longitudinal analysis over the x86 architecture. Among other things, the authors inspect the Linux variability model growth pace, how its structure is affected and which changes developers execute over time.

A sole focus on the configuration space, however, does not provide a full understanding of how variability evolves. In fact, such an analysis can easily lead to wrong conclusions. The variability model of the x86_64 architecture illustrates that: between releases 2.6.32 and 2.6.33, 281 new feature names were added, while 43 were removed. A closer inspection of all spaces of the commits removing such features led us to conclude that 35% of them continued to exist; as our patterns show, developers remove these features from the variability model while migrating them to the implementation side or merging them with other features.[1]

The patterns we present is the first work capturing variability evolution in a multi-space setting of a complex real-world variant rich software. Furthermore, our patterns comprise evolution steps not covered by previous work [4, 5, 12, 14, 8].

We believe that a holistic understanding of evolution practice of complex systems with rich variability will have significant impact on product line research, including work

[1]Renames were also noted.

FOSD'12, September 24–25, 2012, Dresden, Germany.

on methodologies, architectures, modeling languages, automatic analyses and tooling.

The rest of this paper is organized as follows: in Sec. 2 we provide a comprehensive understanding of the three spaces of the Linux kernel, and how they relate to each other. In Sec. 3 we discuss the methodology for extracting our catalog of evolution patterns, which are then presented in Sec. 4. In that section, we show the structure of each pattern, with concrete examples and discussion. We then analyze possible threats to validity of our findings in Sec. 5, and present related work in Sec. 6. We conclude the paper in Sec. 7, along with directions for future work.

2. BACKGROUND

The variability in the Linux kernel appears in three main spaces: (i) configuration space, comprised of Kconfig files; (ii) compilation space: set of kernel build files (KBuild), and; (iii) implementation space: mostly C source code. We present them now in more detail.

Configuration space.

Kconfig is the language in which features and their dependencies are declared. The kernel configurator (xconfig)[2] renders the Kconfig model as a tree of features, from which users select the ones of interest (see Fig. 1). For instance, users interested in a cluster file system can select the OCSFS2 (Oracle™Cluster File System) feature, whose Kconfig snippet is shown in Fig. 2.

Features in Kconfig are mostly written as configs (Fig. 2, lines 3 and 12), and may contain attributes such as type, prompt, dependencies, implied selections, default values, and help text. In our example, OCSFS2 is a tristate feature (line 4): it can be absent (n) or users can select it to be either compiled as a dynamically loadable module (m – shown as a dot in Fig. 1) or statically compiled into the resulting kernel (y – shown as a tick in Fig. 1). Boolean features (line 13) are also possible, assuming either y or n as value. Other types include integer and strings (not shown). A prompt message is a short description of a feature (lines 4 and 13), and it is used by the configurator when rendering the feature in the hierarchy. Features without a prompt are not visible to users. Dependencies (line 5) state a condition that must be satisfied to allow selection of the feature. A select attribute (line 6) enforces immediate selection of target features (CONFIGFS_FS). A default attribute (line 15) states the initial value of a feature, which might later be changed in the configuration process. The feature hierarchy depends on the order in which features are declared and on their dependencies. Cross tree constraints are defined using *select* and *depends on* attributes, but also by default values in combination with visibility conditions. Visibility conditions and default conditions (not shown) are guard expressions over feature names that follow prompt and default attributes: for prompts, it controls whether the feature should be made visible; for defaults, it controls which default attribute is applicable when more than one is defined. For a full mapping from Kconfig to standard FODA feature models, refer to [14, 3]. Formal semantics of Kconfig is presented in [13].

The configurator generates a .config file, which is basically a sequence of (*feature-name*, *feature-value*) pairs. Given the features OCFS2_FS (*OCFS file system support*) and OCFS2_FS_POSIX_ACL (*OCFS POSIX Access Control Lists*) as configured in Fig. 1 results in the following .config snippet:

```
CONFIG_OCFS2_FS=m
...
CONFIG_OCFS2_FS_POSIX_ACL=y
```

[2]Other configurators also exist: config, menuconfig, nconfig, gconfig, etc.

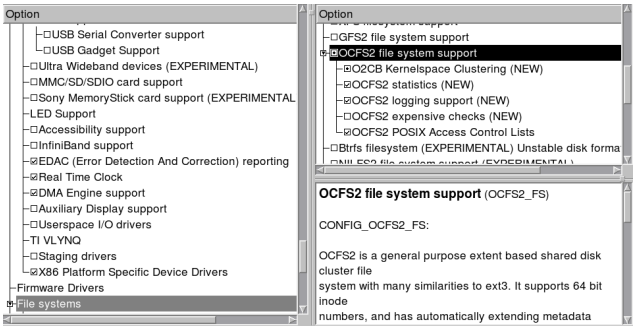

Figure 1: Linux configurator (xconfig)

Compilation space.

The KBuild system controls the compilation process of the Linux kernel. In KBuild, the files containing compilation rules are essentially Makefiles with implicit rules [16]. The image of the kernel is defined by the vmlinux-all goal contained in a top Makefile, whose snippet is shown in the first part of Fig. 3. To build the image, vmlinux-all requires the object files of the symbols appearing at the right hand side of the goal (line 3), which are then linked together. In that case, it requires all the object file names stored in core-y, libs-y, drivers-y and net-y variables. These variables denote lists of object files to which other elements can be appended to. If directories are appended (line 5), KBuild recursively runs the Makefile contained in each such directory and generates one object file per directory based on the content of a special list: obj-y (similarly, a list obj-m controls module compilation). Objects may be conditionally added to this list by replacing y with a feature name. As shown in the second fragment of Fig. 3 (line 3), ocfs2.o is only added to obj-y if the feature OCFS2_FS is set to be y in the .config file. KBuild attempts to compile object files by locating a correspondent C file matching the same name. However, that is not always the case. For ocfs2.o, there is no ocfs2.c file in the Makefile's directory, so KBuild relies on a list named ocfs2-objs (line 11) as the set of object files that should compose ocfs2.o. As before, objects may be conditionally added to such a list (line 10).

Implementation space.

Variability in the source code base is expressed in terms of conditional compilation macro directives, whose conditions are Boolean expression over feature names (see Fig. 4). It is worth noting that before KBuild compiles any code, it reads the content in the .config file and creates an autoconf.h header file containing macro definitions for all features that should be part of the kernel, along with their values. KBuild forces this file to be included in all C sources (this is achieved using gcc's -include switch). For instance, selecting OCFS2_FS_POSIX_ACL for the OCFS2_FS module results in a definition such as

```
# fs/ocfs/Kconfig
...
config OCFS2_FS
        tristate "OCFS2 file system support"
        depends on NET && SYSFS
        select CONFIGFS_FS
        ...
        help
          OCFS2 is a general purpose extent
          based shared disk cluster file system...
...
config OCFS2_FS_POSIX_ACL
        bool "OCFS2 POSIX Access Control Lists"
        depends on OCFS2_FS
        default n
        ...
...
```

Figure 2: KConfig file snippet for OCFS2_FS

top Makefile

```
vmlinux-all := $(core-y) $(libs-y) $(drivers-y) $(net-y)
...
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
...
```

fs/ocfs2/Makefile

```
obj-$(CONFIG_OCFS2_FS) += ocfs2.o ...
ocfs2-objs := ...
        aops.o
        blockcheck.o
        ...
        xattr.o

ifeq ($(CONFIG_OCFS2_FS_POSIX_ACL),y)
   ocfs2-objs += acl.o
endif
...
```

Figure 3: KBuild Makefile snippets

```
// File: fs/ocfs2/acl.h
...
#ifdef CONFIG_OCFS2_FS_POSIX_ACL
extern int ocfs2_check_acl(struct inode *, int);
extern int ocfs2_acl_chmod(struct inode *);
...
#else
#define ocfs2_check_acl NULL
static inline int ocfs2_acl_chmod(struct inode *inode)
{ return 0; }
...
#endif
```

Figure 4: Conditional compilation

```
#define CONFIG_OCFS2_FS_POSIX_ACL 1
```

which guarantees that the code block in lines 4–6 in Fig. 4 will be compiled, instead of lines 8–11.

From the description so far, it is clear that the Linux kernel variability is a three-dimensional space (variability model, Makefiles and C code), and evolutionary changes such as feature addition, removal, split, merge, rename, etc. may affect not a single dimension, but all three. In addition, the three spaces are glued together by referring to feature names as exported in the .config file. Next, we discuss our methodology in extracting evolution patterns.

3. METHODOLOGY

We collected four patterns from a selection of 140 among 220 feature removals from the configuration space in three kernel release pairs of the x86_64 Linux kernel: (v2.6.32, v2.6.33), (v2.6.26, v2.6.27) and (v2.6.27, v2.6.28). Each pattern documents a situation in which the feature is removed from the configuration space, but continues to exist in the software.

Three of our patterns come from the analysis of (v2.6.32, v2.6.33). Our particular interest in v2.6.32 regards to the fact that it is the baseline kernel in Debian 6.0,[3] one of the most mature and popular distributions in the Linux community.

From this initial analysis, we aimed at sequentially diffing release pairs starting from v2.6.26. We fixed such starting point due to incompatibility issues when using newer kernel build infrastructure with older Kconfig and .config files.

While we analyzed and classified all 43 removals in the pair (v2.6.32, v2.6.33), the selection of removals for analysis in (v2.6.26, v2.6.27) and (v2.6.27, v2.6.28) was rather arbitrary. Our main concern was only to capture a pattern that we had not seen before.

Our infrastructure is built on top of the KBuild system, which we extracted from the Linux source code. With it, we parse Kconfig files and compute the set difference of the features in each pair of kernel releases. To facilitate analysis, we also created a relational database containing all feature additions and removals, which are linked with the associated release pair and commit identifier. The records in this database were constructed by parsing all patches in the Linux Git repository.[4]

Our analysis is based on manual inspection over the collected set of commit patches. Since changes can span more than one commit, whenever a patch is insufficient to draw a sound conclusion, we set to recover other commits changing the feature under investigation or any other feature that may affect it (eg.: a parent feature).

4. EVOLUTION PATTERNS

This section presents in detail four evolution patterns in commits found in the Linux kernel repository.

To reduce clutter, we present each pattern in an abstract manner, capturing the changes in each artifact type. Then, we rely on fragments of real artifacts to exemplify the presented concepts, followed by a discussion of the pattern.

We present the first pattern as a basic walk-through to our notation and adapt it as we proceed with presentation.

4.1 Optional feature to implicit mandatory

In this evolution pattern, depicted in Fig. 5, an optional feature F is removed from the feature model, but becomes unconditionally compiled in source code. Its compilation, however, is subject to the presence of F's parent P.

The pattern is presented in two parts, capturing the structure before the change (shown at left) and after it (shown at right). It abstractly documents changes to a fragment of the variability model (rendered in the FODA notation), shown inside a dashed box; the build artifact (B); source code (C), and; the cross-tree constraint formulae (CTC).

Instance.

[3]http://www.debian.org/

[4]git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

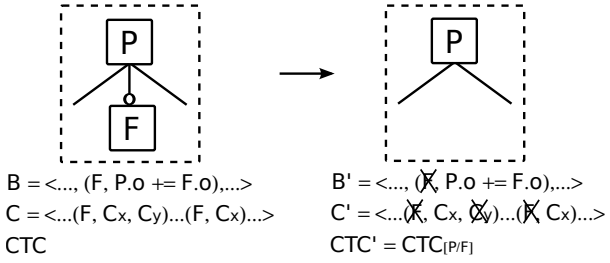

Figure 5: Optional feature to implicit mandatory

```
1  diff --git a/fs/ocfs2/Kconfig b/fs/ocfs2/Kconfig
2  config OCFS2_FS
3  +        select FS_POSIX_ACL
4  -config OCFS2_FS_POSIX_ACL
5  -        bool "OCFS2 POSIX Access Control Lists"
6  -        depends on OCFS2_FS
7  -        select FS_POSIX_ACL
8  -        default n
9  -        help
10 -         Posix Access Control Lists (ACLs) support
11 -         permissions for users...
12
13 diff --git a/fs/ocfs2/Makefile b/fs/ocfs2/Makefile
14 ocfs2-objs := ver.o
15                ...
16                xattr.o
17 -ifeq ($(CONFIG_OCFS2_FS_POSIX_ACL),y)
18 -ocfs2-objs += acl.o
19 -endif
20 +              acl.o
21
22 diff --git a/fs/ocfs2/acl.h b/fs/ocfs2/acl.h
23
24 -#ifdef CONFIG_OCFS2_FS_POSIX_ACL
25  extern int ocfs2_check_acl(struct inode *, int);
26  extern int ocfs2_acl_chmod(struct inode *);
27 ...
28 -#else
29 -#define ocfs2_check_acl NULL
30 -static inline int ocfs2_acl_chmod(struct inode *inode)
31 -{ return 0; }
32 - ...
33 -#endif
34 ...
```

Figure 6: A patch matching the pattern in Fig. 5

The patch[5] fragment in Fig. 6 is a concrete example of this pattern, where OCFS2_FS is P and OCFS2_FS_POSIX_ACL is F. In the patch, changes are either removal (lines prefixed with "-") or addition (lines prefixed with "+"). Lines without any prefix are used as context to ease understanding.

The patch shows that the feature OCFS2_FS_POSIX_ACL is being removed from the feature model (lines 4–11), but its implied selection attribute is moved to its parent feature (line 3). Fig. 5 captures this situation by deleting F from the feature model and by replacing any references to F with P in the set of cross tree constraints, thus leading to a new set CTC'.

Regarding the changes in the Makefile, the patch shows that the compilation condition guarding acl.o is dropped (lines 17–19), and acl.o is unconditionally added to the list of objects ocfs2-objs (line 20). To capture this abstractly, we first introduce a simplified representation for build files. In our notation, build files are denoted as a sequence B of build rules of the form (e, r_1, r_2), where e is a guard expression over feature names (as in line 17 of the patch); r_1 is a build rule in case e evaluates to true; and r_2 is the alternative build rule to be used in case e does not hold. For simplicity, the condition may be omitted (taken as true) to represent unconditional build rules. Moreover, the second rule may not be shown, stating the absence of an alternative rule in case the guard expression fails. Using this notation, we capture the change over the Makefile shown in the patch as follows: in the left side, (F, P.o += F.o) is one build rule in B, stating that if F is present, then F's object code should be part of P's. After the change is applied, a new sequence B' is obtained containing a new build rule where the condition over F is dropped, which we explicitly represent by writing it as crossed:

B' = <..., (~~F~~, P.o += F.o),...>

As for the edits in the source code side (see acl.h: lines 24–33), the patch indicates that the code guarded by a conditional compilation directive is kept, while the associated condition (line 24) and the alternative code block (lines 28–33) are removed. We capture this situation in our abstraction by removing specific parts (shown as crossed) of guarded blocks, which we represent as triples (e, Cx, Cy): similar to build rules, e denotes a conditional macro expression over feature names, whereas Cx is the code to be compiled in case e holds; otherwise Cy is used.

Discussion.

The purpose of this pattern is to guarantee that a security feature is not unintentionally left unselected in face of its parent feature presence; thus, it eliminates the chance of misconfigurations, with the cost of a bigger product (executable binary size). In our example, making Posix Access Control Lists a mandatory feature for the OCFS2 file system is in tune with that: in Linux, ACL controls file/directory permissions for groups and individuals, and it is a major security feature already supported by other filesystems, including ext3/4, xfs, btrfs, etc. In server environments using a cluster based filesystem, it is likely the case that such support is required, and its absence (unintentional or not) might lead to major security flaws, as no permission control would exist.

Interesting enough, users configuring new versions of the kernel in which OCFS2_FS_POSIX_ACL is not available as a selectable feature may conclude that OCFS2 dropped support for ACL. This occurs because the patch removing the OCFS2_FS_POSIX_ACL feature from Kconfig does not update the help text of OCFS2 to state that ACL is now an integral part of it; thus, users might not select OCFS2 as part of the kernel, driven by the conclusion that it now lacks a feature it once supported.

4.2 Computed attributed feature to code

In this evolution pattern, shown in Fig. 7, an invisible feature F (no prompt) is defined by a default expression e.[6] The purpose of F is to be a mere value place holder that is referred in code using the feature's name. The change removes F from the feature model, while replacing its usage in code by its computed default expression. The build artifacts and the set of cross tree constraints are not altered, meaning that F is not referred in constraints and it does not have an

[5]Commit id: e6aabe

[6]Kconfig does not allow arbitrary non-Boolean expressions

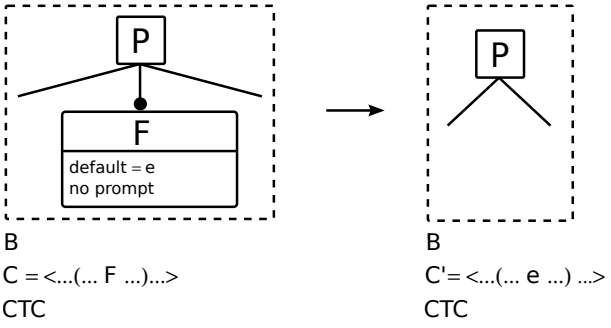

Figure 7: Computed attributed feature to code

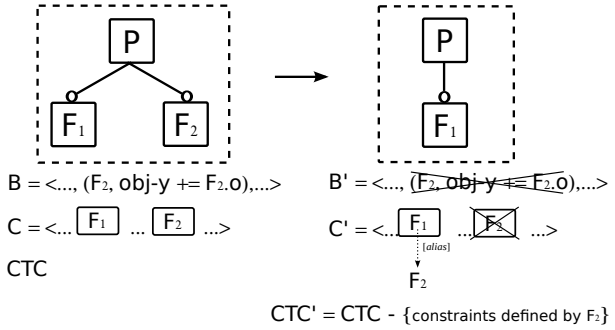

Figure 8: Merge by aliasing

associated compilation unit.

Instance.

An instance of this pattern regards the removal of feature CFG80211_DEFAULT_PS_VALUE[7] (matches F), defined as:

```
config CFG80211_DEFAULT_PS_VALUE
   int
   default 1 if CFG80211_DEFAULT_PS
   default 0
   depends on CFG80211
```

As can be seen, the above definition lacks a prompt message, and thus the feature is not visible to users. Its value is given by a combination of default conditions (refer to Sec. 2), and depends on the presence of CFG80211_DEFAULT_PS. These conditions denote a single abstract conditional expression

```
CFG80211_DEFAULT_PS ? 1 : 0
```

In the source code, the feature is originally referred by

```
rdev->wiphy.ps_default = CONFIG_CFG80211_DEFAULT_PS_VALUE;
```

which was later changed to

```
#ifdef CONFIG_CFG80211_DEFAULT_PS
    rdev->wiphy.flags |= WIPHY_FLAG_PS_ON_BY_DEFAULT;
#endif
```

The inspected patch shows that a set of related Boolean flags in the source code, including ps_default, became a single integer variable (flags) implementing a bit mask. In that sense, the bit-or assignment as shown has the same effect as before, but using a different implementation technique. In case the flag is not set (the conditional statement is not compiled), the corresponding bit position defaults to zero. Otherwise, its associated bit receives 1 as value.

Discussion.

This pattern affects the set of configurations derivable from the configuration space, but it preserves behaviour in all products containing P, as our instance showed. In that sense, the pattern documents a refinement scenario. The existing theory over software product line refinement [5] fails to address this, as its theorems[8] only cover situations with feature model equality or equivalence in the set of possible configurations (our .config files).

Contrary to the previous pattern, this evolution pattern is a refactoring, as it preserves behaviour and improves maintainability, at least as stated by developers in the commit log message:

"We've accumulated a number of options for wiphys which make more sense as flags as we keep adding more. Convert the existing ones."

The choice of having features as place holders for computed attributes in Kconfig files appears to be mere idiomatic preference, as there is no mentioning in the kernel coding style[9] and Kconfig language reference[10] stating which practice is preferable.

4.3 Merge features by module aliasing

This evolution pattern, illustrated in Fig. 8, merges features F_1 and F_2 into the existing feature F_1 when the implementation of F_1 subsumes F_2. The source code comprising the compilation unit of F_2 is completely removed, and so is any build rule. Any constraints defined by F_2 are deleted, and existing constraints remain as is, which means that F_2 is not referred in any other constraint. Furthermore, F_1 registers itself as an alias module to F_2. In that case, whenever the kernel receives a request to load F_2, F_1 is the actual module that gets loaded.

Instance.

An instance of this pattern concerns the merge[11] of the feature RT3090 (matches F_2) into RT2860 (matches F_1), with RT2860 supporting both Ralink™2860 and 3090 wireless chips. In the patch associated with this instance, all the code related to RT3090, its Kconfig entry and build files are removed. The only addition in the patch occurs in rt2860/pci_main_dev.c:

```
+ MODULE_ALIAS("rt3090sta");
```

where rt3090sta is the original object filename created for RT3090, as defined by the rt3090sta-objs list in its Makefile. In the above statement, RT2860 declares that it has RT3090 as its alias.

Discussion.

Merge by alias is only possible for features that are not scattered in code, but rather have a well defined set of files that once compiled generate a single object code.

Contrary to the instance found in *Optional feature to im-*

[7]Commit id: 5be83d

[8]See theorems 11-14 in [5].

[9]http://www.kernel.org/doc/Documentation/CodingStyle

[10]http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

[11]Commit id: e20aea

Feature	Files	SLOC (.c)	SLOC (.h)	SLOC (Makefile)
RT3090	108	56,617	15,318	68
RT2860	88	38,010	10,218	49

Table 1: CLOC statistics for RT3090 and RT2860 drivers

plicit mandatory, the description and help message of the RT2860 feature are updated to reflect the fact that it now supports the RT3090 family of chips.

It appears that RT2860 inherits much of the code from RT3090, suggesting co-evolution of the two drivers. Running the code clone detection tool CCFinder [7][12] supports our claim, as we found 864 clones between the two drivers, with clones containing as many as 2,500 tokens (see Fig. 9 for the whole distribution). Curiously, RT2860 is smaller than RT3090, as we observed by running CLOC.[13] Table 1 shows a reduction of ≈ 32% in SLOC in comparison with RT3090's (.h and .c files), with a Makefile 27% more compact. Despite such a simplification in code, functionality has not been lost, as developers state in the commit log:

"Remove no longer needed rt3090 driver. rt2860 handles now all rt2860/rt3090 chipsets."

In Linux, it is possible to create a single driver supporting multiple devices. This mechanism is also used by developers as a means to merge features. For instance, the driver for the light sensor device TSL2561 is now merged into TSL2563,[14] which supports four devices, as declared in its device table:

```
static const struct i2c_device_id tsl2563_id[] = {
        { "tsl2560", 0 },
        { "tsl2561", 1 },
        { "tsl2562", 2 },
        { "tsl2563", 3 },
        {}
};
MODULE_DEVICE_TABLE(i2c, tsl2563_id);
```

Structurally, this instance is very much related to the instance previously discussed. Its difference relies on how these two features evolved: TSL2563 was implemented completely separate from TSL2561, and was released by Nokia™; TSL2563, on the other hand, was implemented by a single developer. Moreover, the two implementations share no similarity, as CCFinder does not detect any clone between them. This example shows the distributed development nature of Linux, and how drivers released by manufactures tend to subsume drivers developed by the open source community.

4.4 Optional feature to kernel parameter

In this evolution pattern, whose structure is presented in Fig. 10, an optional feature F is removed from the feature model, but continues to exist in the source code. The key aspect of this pattern relies in its build rules. Originally, the presence of a feature F defines a new symbol name (macro) that is appended to the macro namespace of the source code under compilation. Such symbol (X) conditions a block of code S. After the change, F is removed as a feature and it is turned into a kernel parameter F.param that conditions the execution of S during runtime. In that case, the build rule defining symbol (X) is dropped.

[12]ccfx d cpp -dn rt3090 -is -dn rt2860 -w f-w-g+
[13]http://cloc.sourceforge.net/
[14]Commit id: eaacdd

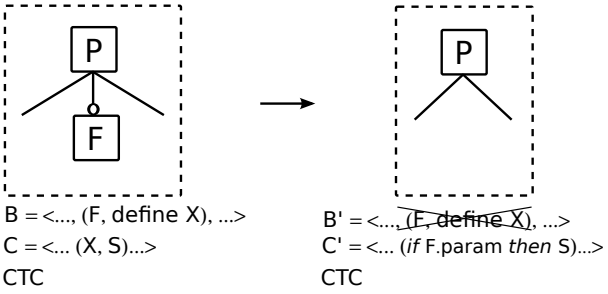

Figure 10: Optional feature to kernel parameter

Instance.

An instance of this pattern concerns the feature CONFIG_PNP_DEBUG,[15] which controls debugging print of Plug and Play devices. Inspecting the Makefile elicits how symbols are appended to the set of defined macro namespace:

```
-ifeq ($(CONFIG_PNP_DEBUG),y)
-EXTRA_CFLAGS += -DDEBUG
-endif
```

As shown above, the GNU C compiler allows macros to be defined through the -D switch. In our instance, the CONFIG_PNP_DEBUG feature was replaced by the boot parameter pnp.debug.

Discussion.

This pattern shows how intricate the Linux kernel three dimensional space is. As illustrated by our instance, the variability switches from being statically compiled to being determined during runtime. Since no functionality is loss and behaviour is preserved, this change results in a software refinement. For the same reasons argued before, evolution occurs in such a way not predicted by existing theory [5].

5. THREATS TO VALIDITY

The major threat to our work is the incompleteness associated with the analysis of commit logs. Our set of inspected commits resulting in features being removed from the configuration space required us to grep associated commits to have a broader picture of the evolution in place. As this process may fail to recover all associated commits, there is a threat that our evolution patterns reflect a partial view of the real changes. This is why we only present the findings as a preliminary sample of patterns. Further experiments will have to broaden the catalog towards completeness and identify whether these patterns are indeed common.

Furthermore, our analysis is ultimately based on the manual inspection over commits to extract the patterns herein presented. As this process contain certain subjectivity, our patterns may not capture the full intention as envisioned by the original patch authors. To alleviate this, we present concrete instances of each pattern to allow readers to judge whether they reflect the presented structure.

6. RELATED WORK

Existing research has already studied the Linux kernel variability. She et al. [14] and Lotufo et al. [8] analyse, among other things, how the Linux variability model evolves

[15]Commit id: ac88a8

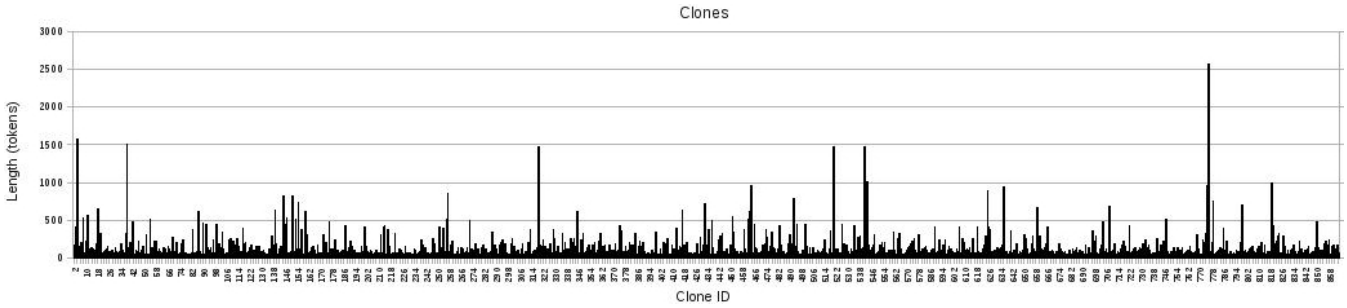

Figure 9: Clones between RT3090 and RT2860 drivers

in terms of feature addition and removal. As we argued in this paper, an analysis based on a single space is incomplete and possibly misleading: features that are no longer present in the variability model do not necessarily cease to evolve, as they might be merged into other features, migrated to implementation space, etc.

Other researchers [5] study the formal aspects of software product line refinement, deriving an evolutionary theory. Such formalism assumes that changes are safe, i.e., do not affect behaviour nor prevents instantiating existing products. Our work shows that Linux does not follow a safe evolutionary model, as certain features are truly removed along the way. Although the authors do not claim completeness, we found real refinement patterns that cannot be explained by their set of theorems.

Borba et al. [4] and Neves et al. [12] provide a catalog of safe transformation templates that, different from ours, do not cover variability evolution when features are removed from the configuration space. In [12], the authors provide evidence on how frequent their templates occur by analyzing the evolution of two small software product lines.

Tartler et al. [17] study inconsistencies in the implementation side by not being kept in synchronization with the variability expressed in Kconfig files. Nadi and Holt [10] identify anomalies in build artifacts, and later extends Tartler's work [11] to detect anomalies across all spaces (configuration, implementation and compilation).

Berger et al. [3] compare Kconfig with other variability modeling languages, such as eCos CDL[16] and standard FODA notation. She and Berger [13] study the semantics of Kconfig and its approximation to propositional logic.

Other studies [1, 2] apply static analysis techniques in Makefiles of Linux and FreeBSD to extract feature-to-code mappings.

7. CONCLUSION

We presented a preliminary catalog of evolution patterns extracted from the Linux kernel repository, and explained each pattern in a comprehensive manner, including (but not restricted to) structure, concrete instances and the mechanisms used by developers in achieving them.

Our study is the first to provide explanations on how variability simultaneously evolves in the implementation, compilation and configuration spaces when removing features from the variability model, while keeping them as part of the software. Furthermore, we rely on a complex and variant rich subject of analysis: the Linux kernel.

[16]sourceware.org

As future work, we aim to execute a longitudinal study of the Linux kernel to assess the frequency of the patterns we found, along with the discovery of new ones. To allow generalization, we plan to perform similar studies in different software product lines, possibly from different domains.

8. REFERENCES

[1] T. Berger, S. She, K. Czarnecki, and A. Wąsowski. Feature-to-Code mapping in two large product lines. Technical report, University of Leipzig, 2010.

[2] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski. Feature-to-code mapping in two large product lines. In *Proceedings of the 14th International Conference on Software Product Lines (SPLC)*, pages 498–499. Springer-Verlag, 2010.

[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 *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 73–82, 2010.

[4] P. Borba. An introduction to software product line refactoring. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, pages 1–26, 2011.

[5] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *Proceedings of the 7th International Colloquium Conference on Theoretical Aspects of Computing (ICTAC)*, pages 15–43, 2010.

[6] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference (SPLC)*, pages 81–90, 2009.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)*, 28(7):654–670, 2002.

[8] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the linux kernel variability model. In *Proceedings of the 14th International Conference on Software Product Lines (SPLC)*, pages 136–150, 2010.

[9] M. Mendonca, M. Branco, and D. Cowan. S.p.l.o.t.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on*

Object Oriented Programming Systems Languages and Applications (OOPSLA), pages 761–762, 2009.

[10] S. Nadi and R. Holt. Make it or break it: Mining anomalies from linux kbuild. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE)*, pages 315–324, 2011.

[11] S. Nadi and R. Holt. Mining kbuild to detect variability anomalies in linux. *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 107–116, 2012.

[12] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the safe evolution of software product lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, pages 33–42, New York, NY, USA, 2011. ACM.

[13] S. She and T. Berger. Formal semantics of the kconfig language. Technical note, University of Waterloo, 2010.

[14] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. In *In Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMos)*, pages 45–51, 2010.

[15] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the linux kernel a software product line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (OSSPL)*, 2007.

[16] R. Stallman, R. McGrath, and P. D. Smith. Gnu make manual, 2010.

[17] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–21, 2012.