Software Product Line Evolution: the Linux Kernel

I. Maria Attarian
Software Engineering Lab
University of Waterloo
Canada
imattari@uwaterloo.ca

Kacper Bąk
Generative Software
Development Lab
University of Waterloo
Canada
kbak@gsd.uwaterloo.ca

Leonardo Passos
Generative Software
Development Lab
University of Waterloo
Canada
lpassos@gsd.uwaterloo.ca

ABSTRACT
Software product lines promote explicit modeling of software variability and systematic reuse of underlying components. Evolution of software product lines occurs both in variability models and assets, such as source code. In this work, we did an empirical study that investigated evolution of the Linux kernel as an example of a product line. We classified different types of edits, and answered questions about congruency of changes made by developers. We also discovered patterns for adding, removing, and updating features in the Linux kernel. As a result, the work presents a set of guidelines for tools that would support evolution of software product lines.

Categories and Subject Descriptors
D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement —*Restructuring, reverse engineering, and reengineering*

General Terms
Restructuring, reverse engineering, and reengineering

Keywords
software product line, evolution, the Linux kernel

1. INTRODUCTION
Software product lines (SPLs) deliver customized software from a shared set of resources, such as code or models. They promote the creation of a *family of software products* from a set of common features[1] instead of the creation of each product from scratch [2].

Software families are often described in terms of *problem space* and *solution space*. Problem space captures high-level requirements, usually in the form of feature models [8], i.e., diagrams in which features are hierarchically arranged and rules on how they can be combined are made explicit. Solution space, on the other hand, contains shared assets, such as source code or models.

Software product lines have been successfully used by several companies in different domains of expertise, such as avionics, automobile industry, telephony, finance, etc. The benefits of its adoption include [16]:

- Improved productivity by as much as 10x.
- Increased quality by as much as 10x.
- Decreased cost by as much as 60%.
- Decreased labor needs by as much as 87%.
- Decreased time to market (to field, to launch) by as much as 98%.
- Ability to move into new markets in months, not years.

In addition to private sectors, software product lines have also succeed in open source projects. The Linux kernel is probably the most prominent example, due to its amount of features (it has more than 6,000 features) and wide-spread use. Using a product line to develop the Linux kernel probably explains why it supports many different architectures, with frequent stable releases over short periods of time. Kernel releases happen every 2-3 months, with each release being a "major" release [7]. The amount of changes per release is also considerable, as seen in Fig. 1.

Many works have studied the design techniques and cost-efficiency of software product lines [14, 1, 4, 11, 10]. However, little is known about the evolution of SPLs. A recent attempt is the empirical study performed by Lotufo et. al. [9], which investigates general characteristics of the evolution of Linux kernel at the problem state.

This paper extends Lotufo's work by analyzing the evolution of the Linux software product line in both problem and solution space. In particular, we aim to answer five research questions:

RQ1 *What does constitute a software product line refactoring?*

[1]A feature can be understood as a piece of functionality in the overall system or a piece of code that can be configured so as to suit the needs of a product in a given SPL.

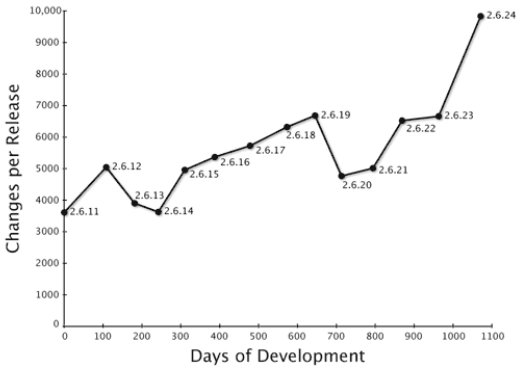

Figure 1: Rate of change per release of the Linux kernel. Extracted from [7].

RQ2 *Are changes to variability model congruent with changes to the source code?*

RQ3 *What kind of tool support would help to evolve a software product line?*

RQ4 *What are the criteria for creating new features?*

RQ5 *Why do developers make some parts of code optional (instead of keeping them mandatory)?*

We also took into account the validation of the SPL refinement theory proposed by Borba et. al. [5]. As they acknowledged, their theoretical framework is missing an empirical part to test its accurateness. In this sense, the Linux kernel, as a representative software product line, is a good candidate for its evaluation. Having said that, we pose a sixth research question that we also aim to answer:

RQ6 *Is the theory of software line refinements adequate to model evolution of the Linux kernel?*

To answer these research questions, we collected a sample of 360 patches from the the Linux kernel source code repository.

This article is organized as follows: in Sect. 2 we enhance our initial definition of problem space, followed by a discussion of solution space in Sect. 3. Next, in Sect. 4 we define some details on the Linux kernel variability model. Section 5 reports our methodology of extracting the sample of edits (changes) that we analyzed. Section 6 explains each category of edit that we took into account, followed by a discussion on Borba's theory. Section 7 presents our findings and Sect. 8 discusses some of their threats to validity. Section 10 concludes the article.

2. PROBLEM SPACE

Problem space captures high-level requirements, usually in the form of feature models. Feature models are tree-like structures that specify commonalities and variabilities within a software family. Figure 2 presents a feature model interpretation of the JFF2S subsystem of the Linux kernel, as

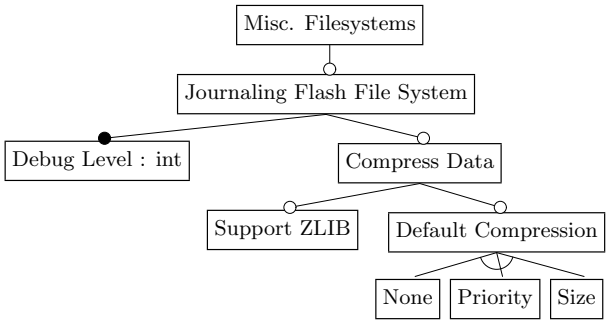

Figure 2: The interpretation of the JFFS2 subsystem feature model. Extracted from [3].

discussed in [3]. Among the possible notation variants [13], referred as FODA-like languages[2], we used the one proposed by Kang et al. [8]. The notation is as follows:

- each box denotes a feature. The top box is the root feature, in our case, the Misc. FileSystems;
- the hierarchy imposed by the tree denotes dependencies;
- an *optional feature* is optionally included in the generated product that contains its parent feature. An optional feature is connected to its parent by means of an edge with a hollow circle. For instance, the Journaling Flash File System is optional – it may or not be present in case the Misc. FileSystems is included in the generated product;
- a *mandatory feature* is always included in all products containing its parent feature. For instance, each time a product is generated with the Journaling Flash File System, the Debug Level feature is also included;
- *alternative features* of a given parent implies that one and only one subfeature will be included given a product containing the parent feature. Alternative features are grouped under a parent feature with a hollow arc. For instance, the features None, Priority and Size are examples of alternative features of Debug Compression;
- *or features* (not shown in the example) are similar to alternative features, except that at least one subfeature must be included in every generated product that contains its parent feature. A set of or-features are grouped under a parent by means of a filled angle.

3. SOLUTION SPACE

Solution space contains shared assets, such as source code or models. There is no one widely-accepted notation for solution space, as it depends on the domain. Common assets have built-in variability, thus they represent a set of products. In many cases this variability is represented by preprocessor annotations, such as IFDEF directives in C or C++. When the user configures the feature model and selects particular features, the selections are reflected in the

[2]FODA: Feature Oriented Domain Analysis.

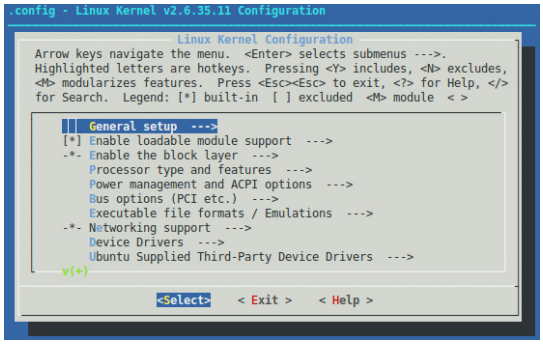

Figure 3: The Linux kernel configurator tool: menuconfig.

source code. As a result, the software product line generates a single program.

A software family covers a set related programs delivered by a software product line. In fact, the set of available configurations defines the semantics of each product line. For example, two software product lines are equal if they generate the same set of products. Sets of configurations change over time, as developers add new features or remove deprecated ones. By knowing what products are available, we can compare software product lines and understand their evolution.

4. THE LINUX KERNEL VARIABILITY

As with other operating systems, the Linux kernel is a software product line. It is a highly variable piece of software that can be tailored to work on a great range of architectures: from personal computers to satellites.

New variants of the Linux kernel can be configured by means of a specialized tool – menuconfig[3]. Building a customized Linux kernel brings some advantages over the use of precompiled ones, such as:

- the possibility of tailoring the kernel towards a specific target architecture;
- load a given driver as part of the kernel or set support for a given driver to be later loaded;
- optimize memory space and boot time by removing useless drivers.

The configurator tool loads the Linux feature model and presents it to the user so it can be tailored accordingly. Figure 3 shows the main screen of the menuconfig tool.

The Linux feature model is a hierarchy of menus and configuration options spread across several different files. These files are coded in KConfig, a domain specific language designed specifically for coding the Linux kernel variability model. It is more expressive than the original feature model notation

[3]menuconfig is a terminal based application. There is also the possibility of using an X-version of it called xconfig.

```
menu "I2C Hardware Bus support"

comment "PC SMBus host controller drivers"

depends on PCI
...

config I2C_NUC900
    tristate "NUC900 I2C Driver"
    depends on ARCH_W90X900
    help
    Say Y here to include support for I2C controller in the
    Winbond/Nuvoton NUC900 based System-on-Chip devices.
...

endmenu
```

Figure 4: A fragment of the I2C Hardware Bus support KConfig file.

defined by [8], but its objective remains the same: represent possible software variants. As a brief introduction to KConfig, its syntax and semantics, consider the fragment files shown in Fig. 4 and Fig. 5, extracted from two of the patches analyzed during our study.

In Fig. 4, the feature I2C Hardware Bus support is declared. The I2C Hardware Bus support feature can only be enabled if the PCI is also enabled, as stated by the depends clause. The configurator will place any configuration option declared inside the I2C Hardware Bus support as its children. In this sense, I2C_NUC900 is a subfeature of I2C Hardware Bus support. The string following tristate sets the menu item name that will be rendered by the configurator tool. The value associated with I2C_NUC900 is a tristate, i.e., it is either zero (disabled), one (enabled) or two (modulized). The modulized option enables the feature to be built as a module that can be added or removed from the kernel once it is loaded. Other possible data types in KConfig include bool, string, hex or int. The I2C_NUC900 feature depends on PCI, inherited from its parent, and on ARCH_W90X900, as defined by its depends clause. KConfig allows cross-dependencies between features and does not restrict them to be only from parents towards children. The I2C_NUC900 feature, for instance, depends on the presence of ARCH_W90X900, which is not its parent feature. The help entry contains the text that will be presented to the user in case he asks for a descriptive help of the given feature.

The second example, presented in Fig. 5, shows the flexibility on how dependency between features can be expressed in KConfig. The HID_APPLE, for instance, is enabled if either USB_HID or BT_HIDP is present. In fact, KConfig allows any sort of boolean expressions to guard the presence condition of a given feature. In KConfig, the selection of a given feature can enable other features. The ZERO_PLUS_FF feature has such characteristic: if enabled, it causes the selection of INPUT_US_FF feature regardless of other constraints.

Once the configuration is performed, the Makefiles are generated properly so the user can invoke the build process. Every enabled feature will have its code compiled. If the feature does not exist as a module in the source code and

```
menu "Special HID drivers"
        depends on HID
...
config HID_APPLE
        tristate "Apple"
        default m
        depends on (USB_HID || BT_HIDP)
        help
        ...
...
config ZEROPLUS_FF
        tristate "Zeroplus based game controller support"
        default m
        depends on USB_HID
        select INPUT_FF_MEMLESS
        help
            ...
endmenu
```

Figure 5: A fragment of the Special HID drivers KConfig file.

instead is tangled with other C code, possibly spread among different C files, its code will be guarded by a conditional compilation macro. As an example, consider the following guarded command that is part of the ZEROPLUS_FF feature code:

```
...
#ifdef CONFIG_ZEROPLUS_FF_MODULE
   HID_COMPAT_CALL_DRIVER(zeroplus);
#endif
...
```

If the user selects ZEROPLUS_FF to be present in the customized kernel, CONFIG_ZEROPLUS_FF_MODULE will assume either 1 or 2 as its value (in C, any value different from zero is taken as true). The guarded code will thus evaluate to true and the enclosed code will be compiled.

5. DATA ACQUISITION

To answer our research questions, we collected a sample of 360 patches from the the Linux kernel Git's public repository [4] and cloned sources of the 2.6 line of the kernel . Although the cloned repository contained the complete history of the kernel, we analyzed only versions 2.6.12 to 2.6.38. Commits belonging to each revision were tagged with a version number, e.g. v2.6.12. We restricted our analysis to such a range because previous versions of the Linux kernel were managed by version control systems other than Git. Developers converted earlier releases to the Git format, but we skipped those releases to assure quality and consistency of the investigated software.

We were interested only in patches that simultaneously modified Kconfig files and source code. We created a script that from a complete list of 232,305 commits extracted only those that modified at least one Kconfig file and at least one C file. Next, the list was randomized using the Unix shell command sort -R. We obtained 7,955 commits and later divided them into 3 non-overlapping pools from which each author analyzed 120 commits. The study involved manual overviewing Kconfig files along with the corresponding implementation. Initially, we expected that manual source code inspection would require more time and effort than analysis of the variability model itself. We discovered, however, certain patterns (e.g. feature addition or deletion) that allowed us to quickly classify majority of commits.

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

6. SPL EVOLUTION

Evolution of software product lines occurs in a series of modifications to problem and solution spaces. Changes happen both in variability models and the underlying assets, such as source code. We looked into the literature [18, 17] and also used 120 training commits from our whole sample to characterize types of modifications. Modifications differ by their impact on software product line semantics.

6.1 Variability Model Edits

Semantics of variability models is specified as a set configurations, i.e. available products. Variability models evolve when corresponding sets of products change. Table 1 shows overlap among existing variability models (empty elipses) and their edited versions (filled elipses). Edits that impact variability models add, remove, or update features or dependencies. Thüm et al. [18] proposed the following set of edits in variability models:

refactoring Preserves the set of products. A refactored variability model generates exactly the same set of products as the original one.

generalization Similarly to refactoring preserves the set of products, but also adds new features or weakens constraints. Consequently, a generalized variability model is a superset of the original one. Generalization is also known as refinement.

specialization Corresponds to feature deletion or adding extra constraints (dependencies). New variability model generates fewer products than the original one. Hence, the old variability model is a generalization of the new one.

arbitrary edit The most general type of edit. At the same time, it adds and deletes products from the original variability model.

The initial analysis of 120 patches provided us confidence on these edit categories in the sense that most of the edits to the Linux variability model were not arbitrary, and thus could be properly classified either as refactoring, generalization or specialization. In fact, only 2 of the 120 edits were considered to be arbitrary.

6.2 Asset Edits

Variability models, such as the Linux kernel Kconfig model, are linked with assets and evolve simultaneously. Changes propagate from problem space to solution space, and the other way around. Semantics of assets defines semantics of solution space, which is usually more complex than semantics of variability models. In the analyzed case it was given mainly by the meaning of C code, i.e. program's behavior and static variability. We classified source code edits as follows:

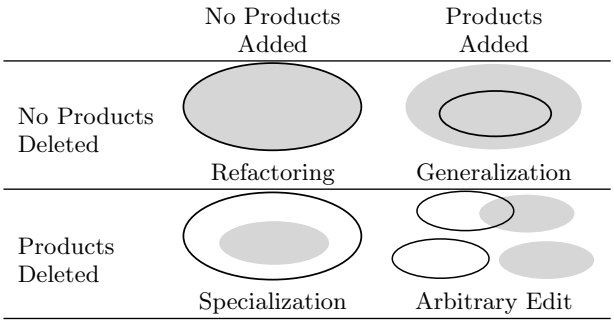

Table 1: Variability Model Edits (from [18])

refactoring Assures behavior preservation. A refactored code has the same functionality but might differ in structure or in nonfunctional attributes.

generalization Preserves behavior of the existing code base and adds extra features to it. New functionality can be brand-new or derived from existing code by making some parts of it optional.

changed component Evolves the code of existing features by adding, removing functionality, or reusing code from other features. The changed functionality does not introduce new variability in the code.

merged components Merges code base of several components. It is done when several components have very similar code that can be generalized and later specialized by parameters. Even though the set of configurations is smaller, available products cover the same functionality; e.g. merging several drivers into one that can handle multiple devices.

specialization Deletes feature from the code base.

arbitrary edit The most general type of edit. Encapsulates all edits that do not fit into one of the above categories.

There is a certain overlap between problem space and solution space edits. Solution space edits could be reduced to the four categories by merging *changed component* and *merged component* with *arbitrary edit*. Yet, we came across the two edits several times: the *changed component* edit was mainly about evolving the component without making changes to variability; the *merged components* edit occurred when developers realized that there are two components that provide very similar functionality.

In theory, updates to problem and solution spaces could be completely independent from each other. Then any combination of variability model and asset edits is possible. However, as will be discussed in Sect. 7, updates to problem space and solution space were always logically related with each other in the set of patches that we analyzed.

6.3 Theory of SPL Refinements

Recent work by Borba et al. presented a general theory of product line refinement [5] that describes evolution of software product lines. Their work considers only generalization edits, i.e. the cases when the set of existing products

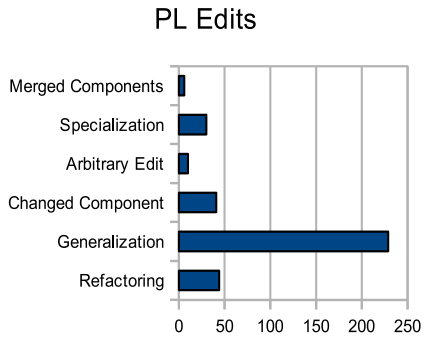

Figure 6: Overall classification of edits in the Linux kernel.

and their behavior are preserved, or the set is extended with new features. Intuitively, a better (generalized) product line is a refinement as long as it can generate enough products to match the original one. It is important, however, that refined products cannot change observable behavior of the original products.

The theory of product line refinement uses set semantics for feature models. As for assets, it abstracts from concrete definition of refinement, but requires asset refinement to be reflexive, transitive, and composable (refining an asset that is part of a valid product yields a refined valid product). The theory defines configuration knowledge as a relation between feature expressions (propositional formulas having feature names as atoms) to sets of asset names. Next, it defines asset mapping as a relation between asset names and assets.

We exercised the theory on the Linux kernel commits to verify its adequateness. Even though Kconfig models are much more expressive than Boolean feature models, Berger et al. [3] presented a translation of Kconfig to feature models. Basing on that translation, and assuming that assets are files or blocks of conditionally compiled code that have implicitly defined unique names, we applied the theory to investigated commits.

7. RESULTS

Through quantitative analysis of our classification of the sample data deriving from the population of patches of the Linux kernel, we extracted four graphs, shown in Fig. 6 to Fig. 8.

RQ1. What does constitute a software product line refactoring?

Refactoring of a software product line acts upon problem space or solution space, or both at the same time. Refactored variability models must preserve the set of available products, while refactored source code cannot change its functionality. We found and analyzed examples where both spaces were modified at the same time, e.g. replacing existing code by part of another feature and adding dependency to the variability model, changing file and variability hier-

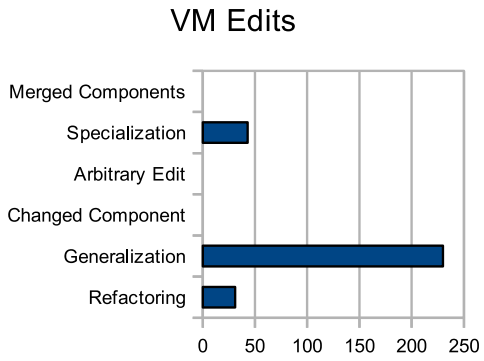

Figure 7: Variability model edits in the Linux kernel.

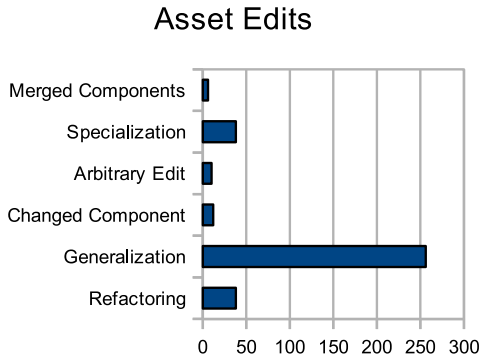

Figure 8: Asset edits in the Linux kernel.

archy, renaming files and functions, removing dead code.

As can be seen in Fig. 6, refactorings are not the most frequent type of edit in the Linux kernel. Most of the times, refactorings are edits that affect source code and the variability model, as can be seen by comparing Fig. 7 with Fig. 8. We found that refactorings that modify variability are most of the times either changes in the name of features or changes in the feature's description, as contained in the help section of a config option. Although not shown by our graphs, renaming is caused first in the variability model and later propagated in the macro definition and their use in C files.

RQ2. Are changes to variability model congruent with changes to the source code?

Investigated commits suggest that Linux is an example of feature-driven development method. Problem space and solution space evolve together, unless developers forgot to reflect changes in either space. Changes to Kconfig referred to relevant part of code base, i.e. functionality was added/removed simultaneously in both spaces. This expected behavior was confirmed in our analysis, as can be observed in by the resemblance of the graphs in Fig. 7 to Fig. 8. In most cases, the type of changes imposed on Kconfig files are congruent with the type of changes observed in the source code, both related to its behavior and its variability.

A similar result was found by Lotufo et. al. [9]. They compared the number of patches added weekly to the Linux source code that modified, and also that did not modify, Kconfig files. At the end, both numbers exhibited almost identical 'heart-beat' patterns, suggesting a causal dependency between changes to the model and to the code (see Fig. 9). Our findings, however, are more specific in the sense that they actually consider the type of edit into account.

We found two ways in which developers introduced new features. Most of the commits introduced new functionality in the form of separate kernel drivers that provide support for new devices. These commits added appropriate entries to Kconfig, Makefiles and new files to the code base. Another way of introducing new features was extraction from existing code and allowing for conditional compilation. In very few cases new functionality added extra dependencies. We noticed that the dependencies did not restrict the original variability model, because they were added to new features.

Some features retire over time and are no longer needed. Analyzed commits showed us that old features are removed, but new features cover their functionality. Again, that was often the case with drivers, when a new more general driver could handle newer and older devices. For this reason when components were merged in the solution space, developers removed redundant variability from Kconfig scripts.

RQ3. What kind of tool support would help to evolve a software product line?

Evolution of a product family requires constant changes to variability model, source code, and links between the two,

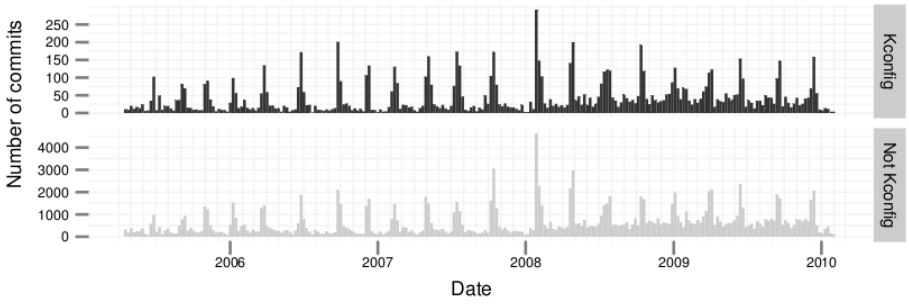

Figure 9: Number of commits of the Linux kernel patches that touched KConfig files versus the number of commits that did not. Extracted from [9].)

i.e. mapping. We found the following common activities that would benefit from automated tool support:

refactoring When a developer changes config name in Kconfig script, then the change must be reflected in source code and Makefiles. Otherwise the configuration and build processes work improperly. Tools should propagate such changes, regardless of whether they originate in Kconfig scripts or source code.

Sometimes developers wish to change physical structure of the code by moving source files. Such a change requires path updates in Kconfig scripts and Makefiles. Furthermore, Kconfig files get deleted and their content is moved somewhere else. Tools should support this refactoring to make the process more transparent and less error-prone.

feature addition Code and Kconfig files evolve together with Makefiles. Tools should update Makefiles' targets when developers add new source files and create new entry in Kconfig scripts.

Another way of introducing new features is extraction from existing code base. Extracted functionality requires updates in Kconfig scripts and adding corresponding Makefiles. Tools should support featurization to convert mandatory code to optional.

feature removal Similarly to feature addition, removing obsolete features results in source code and Makefiles deletion, and in updates to Kconfig scripts.

We also found cases when developers made optional features mandatory, i.e. they removed variability from Kconfig scripts and Makefiles, but did not modify source code behavior. Tools should provide support for converting optional features to mandatory.

updating dependencies The Linux kernel code base is large and developed by thousands of programmers. Sometimes two people provide similar implementations of single functionality. Once they realize that, they update the code so that a single function is shared among several components. That often involves adding Kconfig dependencies among features so that they require feature with the shared function. Developers update references manually, but tools could trace changes in source code, resolve dependencies, and add entries to Kconfig scripts.

RQ4. What are the criteria for creating new features?

Most of the cases observed involving the creation of new features were related to the development of new drivers in order to enable the support of new devices. This is reasonable and was anticipated to some extent. Considering the fast pace of evolution in hardware design and release as well as the large amount of corporations present in the field, the need to provide support to such a widely used operating system such as Linux, and for all these new devices being released to market, increases. Therefore, it was observed that the main motivation of introducing new features to the Linux kernel was the construction of new drivers.

RQ5. Why do developers make some parts of code optional (instead of keeping them mandatory)?

The main reason we identified for making code optional is to give users the right of selection. Since, as mentioned in the previous answer, the devices that can be supported by the Linux kernel are numerous, users should be given the chance to enable/disable their support by the kernel. This allows one to obtain a light weight kernel, loaded with the minimal amount of device drivers and modules, and thus improve performance. The tradeoff between generality and its penalty in performance versus minimality with performance gain is ultimately a user decision.

RQ6. Is the theory of software line refinements adequate to model evolution of the Linux kernel?

The theory works well with our understanding of *generalization* edit, but it breaks when two components are merged into a single component. We assume that the merged component has the same observable behavior as the single components.

According to the theory, assets are uniquely identified by their names. Furthermore, when asset a belongs to the original line and its refined version a' belongs to the refined product line, then their names should be equal. Assuming that there are two different assets a_1 and a_2 belonging to the original line, they must have different names. If a' is a refined asset that covers functionality of both a_1 and a_2, then

its name can only match one of the original names, while it is expected to match both at the same time, which leads to a contradiction.

8. THREATS TO VALIDITY

External. Concerning the external validity of our study, there are also some threats that can be identified. The classes defined in order to perform the classification of the patches investigated, were determined empirically by observation of the type of changes that seemed possible to occur on the variability model and source code of the same set of patches. Therefore, the question of completeness can rise i.e. whether the classes chosen are adequate for the whole set of patches where both the Kconfig file and source code files were altered. Since only a sample of the entire population of such patches was selected for investigation, it is possible that changes imposed on the remaining number of patches might not comply to any of the defined classes in which case a revision of the selected classification would be required. However, the number of patches investigated constitutes an adequate sample specimen according to the laws of statistical analysis. More specifically, it can be calculated that in order to conduct a successful statistical analysis over a population of 7955 elements which is our case, with a confidence level of 95% and confidence interval of approximately 5%, an adequate data sample should contain 360 elements which is the number of patches we manually investigated for this study. Thus, we can make the hypothesis that the classes selected can be considered representative.

It should also be noted that the manual classification of the Linux kernel patches was feasible due to the good organization of the Git repository as well as the clear remarks and comments made by the developers who committed the changes. However, this can be a threat to the validity of the study since this might not be easily generalized. It is not necessary that all or most SPLs are organized in a similar proper manor that would enable the application of the same classification and procedure in order to conclude to valid results.

Also, as it can be seen by the results, a significantly larger number of cases falls into the category of generalization edit. However, many of the conclusions refer to the notion of refactoring when the number of cases classified as refactorings does not constitute a majority. It is, therefore, of question whether the size of the refactoring specimen is adequate to come to valid conclusions concerning the term. To our defence, the overall size of the total sample of patches is compliant with the rules of statistics, as mentioned earlier, and the samples were selected randomly. Thus, the total refactoring cases can be assumed to maintain approximately the same percentage over the overall cases proportionally as observed within the sampled data. Therefore, the sample chosen is likely to be a good indication of the population's behaviour in order to base our results upon. After all, no statistical analysis can result to conclusions with a 100% confidence.

Consequently, it should be noted that only patches modifying both the Kconfig and underlying source code were analyzed for this study. However, it is considered possible that there is a probability of such modifications occurring separately and independently from each other. These cases were ignored for the purposes of this analysis. This can constitute a threat to validity since ignoring such cases can have led to disregarding significant pieces of information which would alter the results of this study.

Finally, the possibility of the Linux kernel not being feature-driven should be mentioned. For the purposes of this study, it was assumed that the Linux kernel is feature-driven i.e. any modifications made to Kconfig files are considered very important aspects of the kernel and its behavior and, therefore, they were expected to be mapped respectively to the source code. However, if this assumption is not valid, this could have led to improper classification choices from our part.

Internal. Concerning the internal validity of this study, there is one threat to validity that ought to be mentioned. As one can observe, the classification of the patches was held manually by taking into account the commentary and the changes into the Kconfig file, underlying source code and Makefile of each commit. Nevertheless, this method lacks a formal set of rules based on which the classification decision is taken. The existence of such a set would provide more credibility and add a sense of formality and objectiveness in the classification procedure. It would, therefore, be of great value if a concrete group of guidelines to be followed in order to make this categorization, could be determined. To our defence, the difficulty of defining such rules should be underlined. While conducting our study, we encountered many cases for which the classification was unclear and arguable. Thus, since there is a certain amount of subjectiveness in the way the changes of each commit are viewed by each reviewer, it can be understood that the definition of a set of rules is not a straightforward and easy task and would not be feasible regarding the time restraints and scope of this project.

The Git repository permits history rewriting to treat cases such as forgotten file additions or improve commenting. Using the diff and log tools of the Git repository, only the latest version is investigated, therefore it is possible that important changes made in previous versions might have been overwritten so are unintentionally ignored by our study. To our defence, investigating the latest version seems more useful since it provides information over the features currently encapsulated in the real product line of the Linux kernel and, thus, reflects changes regarding the present condition of the kernel.

9. RELATED WORK

Recent work on software product line edits focused on feature model updates. Relatively little, however, has been done about classification of edits in the solution space.

Thüm et al. [18] presented an algorithm for automatic classification of variability model edits according to generalization, specialization, refactorings and arbitrary edits. Their algorithm aims to help product line designers measure the impact of their changes on a given feature model. Their main focus regards the correctness of their algorithm and its performance in terms of execution time. The authors do not present any empirical experiments regarding common

patterns of edits in variability models of software product lines, nor any indication of how representative their proposed edit categories are. Our work, on the other hand, gives evince that their proposed work is indeed complete, at least, in what concerns the Linux kernel.

Borba et. al. [5] presented a theory on software product line refinements, which was discussed in Sect. 6.3. Borba's theory has the drawback that it only handles generalizations. Altough the theory is mathematically correct (it was proved using the theorem prover PVS [12]), it is sensitive to cases where two given features are merged, as we noticed by studying the theory itself and the set of patches from the Linux kernel.

Lotufo et al. [9] investigated the evolution of the Linux kernel variability model in terms of 200 randomly selected patches for the x86 architecture. Their sample, in contrast to ours, only considered patches that touched KConfig files. Out of that sample, they categorized modifications into several classes: refactorings, deletions, or arbitrary edits. Their work took into account the number of modified lines in the source code, but disregarded the contents of the changes.

Other product lines resemble the Linux kernel. eCos, for instance, is an open source real time operating system for embedded applications [6]. It has a configurator that different from menuconfig detects conflicts about inconsistencies on a given configuration. As with Linux, it uses a DSL, called Component Description Language (CDL), to express its variability model. Berger et. al. [3] studied the practical use of KConfig and CDL. The authors compared their constructs, semantics, usage and tools, along with the differences with FODA-like languages.

There is no consensus as to whether the Linux kernel is a software product line. Sincero et al. [15] discussed arguments for and against it. There is no doubt that the Linux kernel has artifacts belonging to problem and solution space. An orthogonal dimension to problem and solution spaces is domain versus application engineering. Domain engineering focuses on capturing commonalities and variabilities in a particular domain and on organizing assets for systematic reuse. The Linux kernel development misses domain engineering, although it achieves many goals of software product lines, i.e. configuration, reusability, automatic product derivation. We, however, do believe in Linux as a SPL due to its feature-driven development characteristic.

10. CONCLUSIONS

The purpose of this study was to investigate the notion of evolution in software product lines as well as the way that changes into the variability model affect the source code in real large software product lines. It is expected that in many cases, modifications to feature models should be reflected accordingly into the underlying source code. This attempt of keeping the underlying source code with its feature model in sync poses challenges to software developers which expresses the importance of such an investigation over this subject and, therefore, motivated our work. For the above purpose, the Linux kernel which seems to behave as an SPL and manifests many such characteristics, was chosen as a case study. The conclusions resulting by the statistical analysis held over patches of the Linux kernel can be mainly summed through the following points :

- Refactoring in software product lines is a significant notion although it is not the most common type of alteration met. It concerns all the changes made in the variability model and/or the source code in order for the first to preserve the group of products supported and for the second to maintain the same functionality. Therefore, as expected, in most refactoring cases investigated, both were modified simultaneously.
- Evolution observed in the variability model of the case study was confirmed to affect accordingly the underlying source code, as expected. The addition or removal of features, functionalities and dependencies in the variability model of the SPL, which in this case is manifested through Kconfig files, seems to invoke similar types of changes in the source code in the majority of the samples investigated. Therefore in overall, the variability model is congruent with the underlying source code which generally was assumed necessary to maintain the correct functionality of the system.
- Automated tooling could facilitate the evolution of an SPL. For this purpose, tools should provide support for handling the basic types of changes observed. More specifically, they should enable refactoring, make appropriate changes when new features and functionalities are added or removed and be able to detect and handle dependency updates.
- The addition of new features is the most common case observed. New features mostly concern the support of new devices through the development of driver software. This can be justified if one considers the vast improvement of hardware products, the rapid rate with which they are being released along with the increase in hardware development corporations appearing in the market.

Finally, through this study, previous work was confirmed and additional information was provided concerning software product line refinement and the manor with which evolution is reflected upon the variability models and source code.

11. REFERENCES

[1] F. Bachmann and L. Bass. Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, 26:126–132, May 2001.

[2] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems Journal*, 35, 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 *ASE'10*, 2010.

[4] G. Böckle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid. A cost model for software product lines. In *Software Product-Family Engineering*, volume

3014 of *Lecture Notes in Computer Science*, pages 310–316. Springer Berlin / Heidelberg, 2004.

[5] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *Theoretical Aspects of Computing - ICTAC 2010*, 2010.

[6] eCos. Embedded configurable operating system. http://ecos.sourceware.org/, 2010.

[7] G. K.-H., J. Corbet, and A. McPherson. The linux foundation: The linux kernel development. http://ldn.linuxfoundation.org/article/linux-kernel-development-april-2008, 2008.

[8] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, CMU, 1990.

[9] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the linux kernel variability model. In *SPLC'10*, 2010.

[10] P. Mohagheghi and R. Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans. Softw. Eng. Methodol.*, 17:13:1–13:31, June 2008.

[11] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer, 1st edition edition, 2005.

[12] PVS. Pvs specification and verification system. http://pvs.csl.sri.com/, 2010.

[13] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks Journal*, 51, 2007.

[14] D. Sharp. Reducing avionics software cost through component based product line development. 1998.

[15] J. Sincero, H. Schirmeierand, W. Schröder-Preikschat, and O. Spinczyk. Is the linux kernel a software product line? In *SPLC-OSSPL'07*, 2007.

[16] Software Enginnering Institute. Catalog of software product lines. http://www.sei.cmu.edu/productlines/, 2010.

[17] M. Svahnberg and J. Bosch. Evolution in software product lines. In *3rd International Workshop on Software Architectures for Products Families (IWSAPF-3)*, 2000.

[18] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE'09*, 2009.