Extracting Feature Model Changes from the Linux Kernel Using FMDiff

Nicolas Dintzner
Software Engineering Research Group
Delft University of Technology
Delft, Netherlands
N.J.R.Dintzner@tudelft.nl

Arie Van Deursen
Software Engineering Research Group
Delft University of Technology
Delft, Netherlands
Arie.vanDeursen@tudelft.nl

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

ABSTRACT

The Linux kernel feature model has been studied as an example of large scale evolving feature model and yet details of its evolution are not known. We present here a classification of feature changes occurring on the Linux kernel feature model, as well as a tool, FMDiff, designed to automatically extract those changes. With this tool, we obtained the history of more than twenty architecture specific feature models, over ten releases and compared the recovered information with Kconfig file changes. We establish that FMDiff provides a comprehensive view of feature changes and show that the collected data contains promising information regarding the Linux feature model evolution.

Categories and Subject Descriptors

D.2.9 [**Software engineering**]: Management—*Software configuration management*; D.2.13 [**Software engineering**]: Reusable Software—*Domain engineering*

General Terms

Management,Experimentation

Keywords

software product line, feature model, evolution

1. INTRODUCTION

With more than 10,000 features and decades of development history, the Linux kernel is a popular choice of system for the study of large scale software product line evolution. To understand how such a large and variable system evolves, researchers have been looking at the evolution of its feature model (FM). Several studies (e.g. [7],[5]) quantified the addition and removal of features in the Linux kernel over time or present structural metrics of the kernel's FM, such as the depth of feature structure or number of leaf features in each release, as means to illustrate the evolution of both the kernel and its FM.

VaMoS '14 January 22 - 24 2014, Sophia Antipolis, France
 978-1-4503-2556 ...$15.00.
http://dx.doi.org/10.1145/2556624.2556631.

Previous work on the evolution of FMs of other systems (e.g. [4], [19], and [9]) mention types of changes that have not been studied on the Linux kernel, as attribute value changes for instance. With that in mind, we think that a more detailed view of the changes occurring on the Linux kernel FM can be obtained, thus providing more insights on the evolution of the kernel itself, its architecture, implementation and build mechanism.

In this paper, we present a classification of feature changes occurring in the Linux kernel FM based on the Kconfig language[1] and a corresponding tool, `FMDiff`, to extract them. Our classification describes feature changes on three different levels of granularity. The first level, the coarsest, allows us to capture changes on the level of FMs, namely the addition, modification, and removal of features. The second level describes changes in the properties of features, such as adding a default value to an existing feature, and the third, the finest, reflects changes in attribute values, as the addition of a feature reference in the condition of a select statement for instance. Inspired by `Undertaker` ([21],[20]) and based on the EMF Compare[2] diff algorithm, `FMDiff` captures type, prompt, default value, select, and depends statement changes, as defined by the Kconfig language. Our tool loads and compares Linux FMs extracted from two subsequent versions of the Linux kernel and stores differences (i.e. feature changes) in a database for further analysis.

Using `FMDiff` and our change type classification, we build a dataset comprised of features changes obtained by extracting the change history of more than twenty architecture-specific FMs over ten releases of the Linux kernel.

To evaluate our approach, we randomly extract a set of feature changes from our dataset and manually trace them back to Kconfig file modifications, and vice versa. The results show that `FMDiff` captures a large majority of changes operated on Kconfig files and provides a more comprehensive view of feature changes than what could be obtained by looking at Kconfig file differences. In addition, we present a first statistical view of the changes captured by `FMDiff`, underlining the potential usefulness of detailed information about feature changes when analyzing FM evolution.

[1] https://www.kernel.org/doc/Documentation/kbuild-/kconfig-language.txt
[2] http://www.eclipse.org/emf/compare/

The main contributions of this paper are: 1) the feature model change type classification scheme; 2) the FMDiff approach to extract and classify changes; 3) the evaluation of the classification scheme and FMDiff with ten recent releases of the Linux kernel feature model; and 4) a first statistic analysis on the evolution of the Linux kernel feature model. Finally, FMDiff and our dataset are available for download.[3]

The remainder of this paper is organized as follows. Section 2 provides some background information of the Linux kernel FM and its reconstruction. We present our feature change classification and its rationale in Section 3. FMDiff is introduced in Section 4 and evaluated in Section 5. We discuss the tool capability to capture feature changes and its potential usage in Section 6. Section 7 presents related work. Finally, we conclude this paper and elaborate on potential future applications of FMDiff in Section 8.

2. BACKGROUND

The approach described in this paper is based on the extraction of FMs declared with the Kconfig language using the kdump tool. In this section, we present the basic Kconfig concepts and the output format generated by kdump.

2.1 The Linux kernel feature model

Linux users can tailor their own kernel with Xconfig (among other tools), the kernel configurator. This tool displays available configuration options in the form of a tree. In the following we refer to options as features. As the user selects or unselects features, Xconfig updates the tree of visible features to show only those that are compatible with the current selection. Features and their composition rules, that denote cross-tree constraints in FMs, are specified using the Kconfig language (see also [16] and [17]). Listing 1 depicts an example of a feature declaration in the Kconfig language.

Listing 1: Example of feature declaration in Kconfig

```
1  if ACPI
2
3  config ACPI_AC
4    tristate "AC Adapter"
5    default y if ACPI
6    depends X86
7    select POWER_SUPPLY
8    help
9      This driver supports the AC Adapter
10     object,(...).
11
12 endif
```

In the Kconfig language, features have at least a name (following the config keyword on line 3) and a type. The type attribute specifies what kind of values can be associated with a feature. A feature of type boolean can either be selected (with value y for yes) or not selected (with value n for no). Tristate features have a second selected state (m for module), implying that the features are selected and are meant to be added to the kernel in the form of a module. Finally, features can be of type integer (int or hex) or type string. In our example the ACPI_AC feature is of type tristate (line 4). Features can also have default values, in our example the feature is selected by default (y on line 5), provided that the condition following the default if keyword is satisfied. The text following the type on line 4 is the prompt attribute. It defines whether the feature is visible to the end user during the configuration process. The absence of such text means the feature is not visible.

Kconfig supports two types of dependencies. The first one represents pre-requisites, using the depends (or depends on) statement followed by an expression of features (see line 6). If the expression is satisfied, the feature becomes selectable. The second one, expressing reverse-dependencies, are declared by the select statement. If the feature is selected then the target of the select will be selected automatically as well (POWER_SUPPLY is the target of the select statement on line 7). The select statement may be conditional. In such cases, an if statement is appended to the select statement. depends, select and constrained default statements are used to specify the cross-tree constraints of the Linux kernel FM [14]. A feature can have any number of such statements.

Furthermore, Kconfig provides statements to express constraints on sets of features, such as the if statement shown on line 1. This statement implies that all features declared inside the if block depend on the ACPI feature. This is equivalent to adding a depends ACPI statement to every feature declared within the if block.

Finally, Kconfig offers the possibility to define feature hierarchy using menus and menuconfigs. Those objects are used to express logical grouping of features and organize the presentation of features in the kernel configurator. Like "if" statements, constrains defined on menus and menuconfigs are applicable to all elements within.

2.2 Feature model representation

A prerequisite to our approach is to be able to extract feature definitions from Kconfig files. For this, we use an existing Linux tool, namely kdump, to translate Kconfig features into an easier to process format. This tool has been used in other studies of the Linux variability model, such as [21], where kdump output is used by Undertaker to determine feature presence conditions. kdump produces a set of ".rsf" files, each one containing an architecture specific FM, i.e. an instance of the Linux FM where the choice of hardware architecture is predetermined. Listing 2 shows the example of the feature declared in Listing 1 in ".rsf" format as output by kdump.

Listing 2: Representation of the feature declaration of Listing 1 in .rsf format

```
Item ACPI_AC tristate
Prompt ACPI_AC 1
Default ACPI_AC "y" "X86 && ACPI"
ItemSelects ACPI_AC POWER_SUPPLY "X86 && ACPI"
Depends ACPI_AC "X86 && ACPI"
```

The first line shows the declaration of a feature (Item) with name ACPI_AC and type tristate. The second line declares a prompt attribute for feature ACPI_AC and its value is set to true (1). The third line declares the default value of the ACPI_AC feature, which is set to y if the expression X86 && ACPI evaluates to true. Line 4 adds a select statement reading when ACPI_AC is selected the feature POWER_SUPPLY is selected as well, if the expression X86 && ACPI evaluates to true. Finally, the last line adds a cross-tree constraint reading feature ACPI_AC is selectable (depends) only if X86 && ACPI evaluates to true.

[3]https://github.com/NZR/Software-Product-Line-Research

kdump eases feature extraction but modifies their declaration. Among the applied modifications, two are most important for our approach: kdump flattens the feature hierarchy and it resolves features depends statements. Concerning the flattening of the hierarchy, kdump modifies the depends statement of each feature to mirror the effects of its hierarchy. For instance, kdump propagates surrounding if conditions to the depends statements of all features contained in the if-block. This explains the addition of ACPI to the condition of the depends statement on line 5 of Listing 2. Concerning the resolution of depends statements, kdump propagates conditions expressed in the depends statement of a feature to its default and select conditions. This explains the condition X86 && ACPI that has been added to the select (ItemSelects) and default value (Default) statements.

3. CHANGE CLASSIFICATION

We aim at classifying feature changes occurring on the Linux kernel FM. Existing feature change classifications do not take into account some specificities of the Kconfig grammar (e.g. select relationships with conditions). To capture as accurately as possible changes in such statements, we introduce a new classification. We present a three level classification scheme of feature changes, namely change category, change sub-category and change type. Each category describes a feature change on a different level of granularity. Items in each level are named based on the modified Kconfig statement, such as a default statement, and the change operation applied. The possible change operations that we consider are are add (ADD), remove (REM), and modify (MOD). Figure 1 depicts our change classification scheme.

The first level, change category, describes changes at a FM level. Here, features can be either added, removed, or modified. The corresponding change categories are ADD_FEATURE, REM_FEATURE, and MOD_FEATURE. In the following, we abbreviate lower-level change types by prefixing the feature property that can change with the three change operations ADD, REM, and MOD.

The next level, change sub-category, describes which property of the feature changed. We differentiate between attribute changes (i.e. type or prompt properties), and changes in the dependencies, default value, and select statements. The corresponding twelve change sub-categories are {ADD,REM,MOD}_ATTR, {ADD,REM,MOD}_DEPENDS, {ADD,REM,MOD}_DEF_VAL, and {ADD,REM,MOD}_SELECT.

Finally, change types detail which attribute, or which part of a statement is modified. The change types are:

- Attribute change types: we track changes occurring on the type and prompt attributes. Combined with the three possible operations, we have {ADD,REM,MOD}_TYPE and {ADD,REM,MOD}_PROMPT.
- Depends statement change types: depends statements contain a boolean expression of features. We use a set of change types describing changes occurring in that expression, namely {ADD,REM,MOD}_DEPENDS_EXP. In addition, we further detail these changes by recording the addition and removal of feature references (mentions of feature names) in the boolean expression with the two change types {ADD,REM}_DEPENDS_REF.
- Default statement change types: default statements are composed of a default value and a condition. Both, the condition and the value can be boolean expressions of features. Default values can be either added or removed recorded as {ADD,REM}_DEF_VAL change types. Changes in the default statement condition are stored as {ADD,REM,MOD}_DEF_VAL_COND. Finally, we track feature references changes in the default value using {ADD,REM}_DEF_VAL_REF and in the default value condition using change types {ADD,REM}_DEF_VAL_COND_REF.
- Select statement change types: select statements are composed of a target and a condition which, if satisfied, will trigger the selection of the target feature. Similar to the default statement change types, we record {ADD,REM,MOD}_SELECT_TARGET changes. Changes to the select condition are recorded as {ADD,REM,MOD}_SELECT_COND. Finally, to track changes in feature references inside a select condition, we use the {ADD,REM}_SELECT_REF change types.

The three change categories, twelve change sub-categories and twenty seven change types form a hierarchy allowing us to classify changes occurring in FM declared with Kconfig.

As an example consider an existing feature with a default value definition to which a developer adds a condition. The change will be fully characterized by the change category MOD_FEATURE and the sub-category MOD_DEF_VAL, since the feature and default value declaration already existed, and finally the ADD_DEF_VAL_COND change type denoting the addition of a condition to the default value statement, and a ADD_DEF_VAL_REF change type for each of the features referenced in the added default value condition.

Kconfig provides several additional capabilities, namely menus to organize the presentation of features in the Linux kernel configurator tool, range attribute on features and options such as env, defconfig_list or modules. We do not keep track of menu changes, but we do capture the dependencies induced by menus. kdump propagates feature dependencies of menus to the features a menu contains in the same way it propagates if block constraints. kdump does not export the range attribute of features, therefore we cannot keep track of changes on this attribute and do not include them in our feature change classification scheme. We plan to address this issue in our future work. Furthermore, kdump does not export options such as env, defconfig_list or modules and we cannot track changes in such statements. But, because those options are not properties of features and do not change their characteristics, we consider the loss of this information as negligible when studying FM evolution.

Regarding our classification scheme, we would like to point out that some combinations of change category, sub-category, and change types are not possible or do not occur in practice. For instance, the change types denoting that a depends or a select statement was added can not occur together with the change category REM_FEATURE denoting that the feature declaration was removed. Some combinations are also constrained by Kconfig, such as the change type ADD_TYPE can only occur in the context of a feature creation, i.e. with the change category ADD_FEATURE.

Furthermore, our change classification does currently not include high-level FM transformations, such as merge feature or move feature. However, the effect of such transformations on features can be represented by modifications of feature dependencies which are covered by our classification.

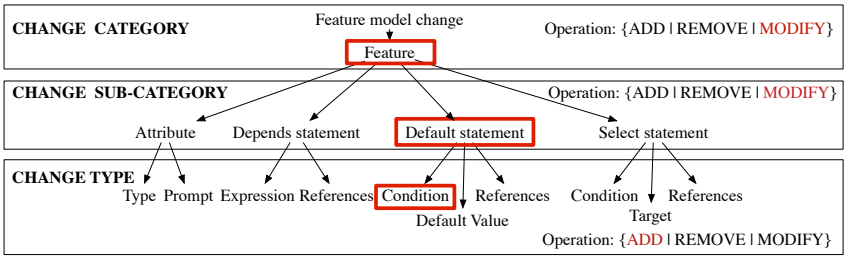

Figure 1: FMDiff scheme to classify changes in feature models on different levels

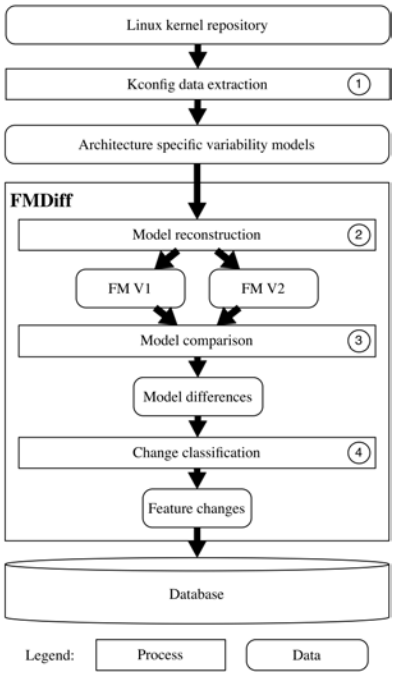

Figure 2: Change extraction process overview

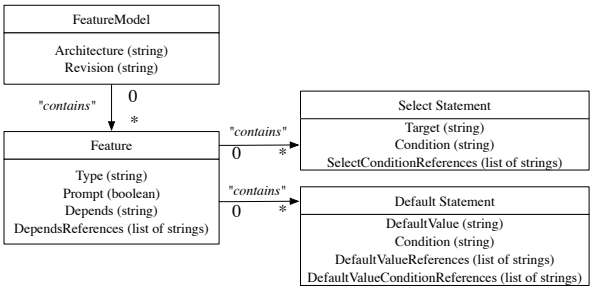

Figure 3: FMDiff feature metamodel

4. FMDIFF

The main objective of `FMDiff` is to automate the extraction of changes occurring on the Linux FM and record those changes according to the classification scheme presented in the previous section. The extraction of feature changes is done in several steps as depicted in Figure 2.

4.1 Feature model extraction

The first step of our approach consists in extracting the Linux FM from Kconfig files. We first obtain the Kconfig files of selected Linux kernel versions from the source code repository. Next, we use the `Undertaker` tool, that provides a wrapping of `kdump`, to extract architecture specific FMs for each version. Undertaker outputs one ".rsf" file per architecture per revision, in the format described in Section 2.

We perform a few noteworthy transformations when loading ".rsf" files into FMDiff. ".rsf" files describe Kconfig choice structures. Those entities are not named in the Kconfig files and are automatically named by kdump (e.g. `CHOICE_32`). This means that the same choice structure can have different names in different releases and cannot be accurately tracked over time. For the time being, we ignore choices when instantiating an FM.

Features can declare dependencies on those `choice`, referring to them by their generated name. We replace all choice identifiers in feature statements by `CHOICE`. Doing this, we cannot trace the evolution of choice structures but prevent polluting the results with changes in the choice name generation order while we still are able to track changes in feature dependencies on choices.

4.2 FMDiff feature model construction

As a second step, we reconstruct FMs from two subsequent versions of a ".rsf" file. FMDiff compares FMs that are instances of the meta-model presented in Figure 3.

`FeatureModel` represents the root element having two attributes denoting the architecture and the revision of the FM. A FeatureModel contains any number of features represented as `Feature`. Each feature has a name, type (boolean, tri-state, integer, etc.), and prompt attribute. In addition, each feature contains a `Depends` attribute representing the `depends` statements of a Kconfig feature declaration. All features referenced by the `depends` statement are stored in a collection of feature names, called `DependsReferences`.

Each feature can have any number of `Default Statements`, containing a default value and its associated condition. Furthermore, a feature can have any number of `Select Statements` containing a select target and a condition. The condition of both statements is recorded as string by the attribute `Condition`. The features referenced by the condition of each statement are stored in the collection `DefaultValueReferences` or `SelectReferences` respectively.

The .rsf output also allows a feature to have multiple `depends` statements but, in our meta-model, we allow features to have only one. In the case where FMDiff finds more than one for a single feature, it concatenates those statements

using a logical `AND` operator. This preserves the Kconfig meaning of multiple `depends` statements.

It is possible for a feature to have two default value statements, with the same default value ("y" for instance) but with different conditions. In such cases, our matching heuristic would be unable to distinguish between the two. The same is true for features that have two select statements with the same target. To circumvent this problem, we concatenate conditions of default statements with a logical `OR` operator if their respective default values are the same. We do the same transformation for select statement conditions, for the same reasons.

4.3 Comparing models

For the comparison of two FMs, FMDiff builds upon the the `EMF Compare` framework. EMF Compare is part of the Eclipse Modeling Framework (EMF) and provides a customizable "diff" engine to compare models. It was used in the past to compare models in various domains, like interface history extraction [12], or IT services modeling [3], and proved to be flexible and efficient. EMF Compare takes as input a meta-model, in our case the meta model presented in a Figure 3, and two instances of that meta-model each representing one version of an architecture specific Linux FM. EMF Compare outputs the list of differences between them.

The diff algorithm provided by EMF Compare is a two step process. The first step, the "matching" phase, identifies which objects are conceptually the same in the two instances. In our case study, this means matching a feature from one FM to the other. Here, we consider two features to be the same if they have the same name in the two models. Similarly, we need to provide rules to identify whether two default or select statements are the same. For default value statements, we use a combination of the feature name and the default value. For select statements, we use the targeted feature name and the feature name.

Our choices of matching rules have consequences on how differences are computed. A renamed feature cannot be matched in two models using our rules. Its old version will be seen as removed, and the new one as added. Default or select statements can only be matched if their associated feature and its default value (or select target respectively) are the same in both models. Changes in default values (select target) are captured as the removal of a default value (select) statement and the addition of a new one.

EMF Compare generates a list of the differences between the two models, expressed using concepts from the FMDiff feature meta-model. For instance, a difference can be an "addition" of a string in the `DependsReferences` attribute of a feature. Another example is the "change" of the `Condition` attribute of a `Select Statement` element, in which case EMF Compare gives us the old and new attribute value.

4.4 Classifying changes

The last step of our process consists in translating the differences obtained by EMF Compare into feature changes as defined by our classification scheme. The translation process comprises four steps. First, we run through differences pertaining to the "contains" relationship of the `FeatureModel` object to identify which features have been added and removed, giving us the feature change category. Then, we focus on differences in "contains" relationships on each `Feature` to extract changes occurring at a statement level, providing us with the change sub-category. The differences in attribute values of the various properties are then analyzed to determine the change type. Finally, changes are regrouped by feature name, creating for each feature change the 3-level classification.

The results are stored in a relational database. We record for each feature change: the architecture and revision of the FM in which the change occurred, the name of the feature affected, the change classification, and the old and new values of the attribute (when relevant).

5. EVALUATING FMDIFF

`FMDiff`'s value lies in its ability to accurately capture changes occurring on the Linux FM. To assess both `FMDiff`'s data completeness and usefulness, we compare it with the information on changes that we obtained by manually analyzing the textual differences between two versions of Kconfig files. We consider `FMDiff` data to be complete if it contains all changes seen in Kconfig files, and useful if it provides the information needed to understand the changes in the Linux FM.

5.1 Data set

Using Git, we retrieve the history of the Linux FM. Lotufo et al. highlight that at random points in time, the Linux FM is not necessarily consistent[7]. To minimize such issues, we extract feature changes between official Linux releases. For all releases of the Linux Kernel from 2.6.28 to 3.8, we rebuild 26 architecture specific FMs. We extract the changes occurring in 10 releases, over time period of 2 years (from March 2011 for 2.6.38 to February 2013 for 3.8).

Between release 2.6.38 and 3.8, 5 new architectures were introduced (*Unicore32* in 2.6.39, *Openrisc* in 3.1, *Hexagon* in 3.2, *C6X* in 3.3 and *arm64* in 3.7). We include those architectures in our study to capture the effects of the introduction of new architectures on the Linux FM. We extract the feature history of 21 architectures present in version 2.6.38 and follow the addition of new architectures, for a total of 26 in 3.8. Our dataset contains 1,860,311 feature changes.

5.2 Completeness

To evaluate the completeness of the captured changes, we verify that a set of feature changes observed in Kconfig files are also recorded by `FMDiff`.

5.2.1 Method

We randomly pick twenty five Kconfig files from different sub-systems (memory management, drivers and so on) modified over five releases. We then use the Unix "diff" tool to manually identify the changed features.

Because `FMDiff` captures feature changes per architecture, we first determine in which architecture(s) those feature changes are visible. Then, we compare Kconfig files diff' with the feature changes captured by `FMDiff` for one of those architectures. We pick architectures in such a way that all architectures are used during the experiment.

For each feature change, `FMDiff` data 1) *matches* the Kconfig modification if it contains the description of all feature changes - including attribute and value changes; 2) *partially matches* if `FMDiff` records a chage of a feature but that change differs from what we found out by manually analyzing the Kconfig files; 3) *mismatches* if the change is not captured by `FMDiff`.

A *partial* or *mismatch* would indicate that FMDiff misses changes, hence the more *full matches* the more complete `FMDiff` data is. We also take into account that renamed features will be seen in `FMDiff` as "added" and "removed".

5.2.2 Results

In the selected twenty five modified Kconfig files, 51 features were touched. 48 of those feature changes could be *matched* to `FMDiff` data, described by 121 records of our database. A single *partial match* was recorded, caused by an incomplete ".rsf" file. A default value statement (*def_bool_y*) was not translated by `kdump` in any of the architecture specific ".rsf" files. In two cases, the `FMDiff` changes *did not match* the Kconfig feature changes. In both cases, developpers removed one declaration of a feature that was declared multiple (2) times, with different default values, in different Kconfig files. In `FMDiff`, a change in the feature default value was recorded, which is consistent with the effect of the deletion on the architecture specific FM. Based on this, we argue that `FMDiff` accurately captures the change.

Over our sample of feature changes, `FMDiff` did capture all the changes occurring in ".rsf" files. Moreover, a large majority (94%) of Kconfig file changes were reflected in `FMDiff`'s data. In the remaining cases, `FMDiff` still captures accurately the effects of Kconfig file changes on Linux FM. We conclude, based on our sample, that the dataset obtained with `FMDiff` is complete with respect to the changes occurring on the Linux FM.

5.3 Usefulness

By comparing `FMDiff` data with Kconfig file differences, we identify what information made available by `FMDiff` would be difficult to obtain using textual difference approaches.

5.3.1 Method

We trace 100 feature changes randomly selected from the `FMDiff` dataset to the Kconfig file modifications that caused them. For each change, we determine the set of Kconfig files of both versions of the Linux FM that contain the modified feature. We then perform the textual diff on these files and manually analyze the changes. If the diff cannot explain the feature change recorded by `FMDiff`, we move up the Kconfig file hierarchy and analyze the textual differences of files that include this file via the `source` statement.

The comparison between `FMDiff` changes and Kconfig file changes can either 1) *match* if the change can be traced to a modification of a feature in a Kconfig file; 2) *indirectly match* if the change can be explained by a Kconfig file change but the feature or attribute seen as modified in the Kconfig file is not the same as the one observed in `FMDiff` data; or finally 3) *mismatch* if it cannot be traced to a Kconfig file change.

We observe an *indirect match* when a `FMDiff` change is the result of `kdump` propagating dependency changes onto other feature attributes or onto its subfeatures (e.g. when a `depends` statement is modified on a parent feature) . Here, *indirect matches* indicate that `FMDiff` captures side-effects of changes operated on Kconfig files.

5.3.2 Results

Among the hundred randomly extracted changes, four were modifications of feature boolean expressions, adding or removing multiple feature references. We traced each reference addition/removal separately, bringing us to a total of 108 tracked feature changes.

We successfully traced 107 changes out of 108 back to Kconfig files changes. A single *mismatch* was found, involving a choice statement couldn't be explained; but the change was consistent with the content of `kdump`'s output. We obtained 26 *matches*, 79 *indirect matches* and finally 2 features were renamed and those changes were successfully captured as deletion and creation of a new feature. Among the *indirect matches*, 61 are due to hierarchy expansion and 18 due to `depends` statement expansion on other attributes.

The large number of indirect matches is explained by an over-representation in our sample of changes induced by the addition of new architectures. Architectures are added by creating, in an architecture-specific folder (e.g. */arch*), a Kconfig file referring existing generic Kconfig files in other folders (e.g. */drivers*). Hence, we observe feature additions in an architecture specific FM without modifications to feature declarations.

79 feature changes captured by `FMDiff` could not be directly linked to feature changes in Kconfig files but to changes in the feature hierarchy or other feature attributes. We argue that even if `FMDiff` data does not always reflect the actual modifications performed by developers in Kconfig files, it captures the effect of the changes on the Linux FM. In fact, `FMDiff` data provides more information than what can be obtained from the textual differences between two versions of the same Kconfig file, where such effects need to be reconstructed manually.

6. DISCUSSION

During our evaluation, we showed that `FMDiff` captures accurately a large majority of feature changes in the Linux FM. Based on this, we elaborate here on limits and potential usages of the tool and the gathered data.

6.1 Feature changes

Thanks to `kdump` hierarchy and attribute expansion, `FMDiff` not only captures changes visible in Kconfig files, but also the side effects of those changes (*indirect matches*). It makes explicit FM changes that would otherwise only be visible by manually expanding dependencies and conditions of features and feature attributes. Such an analysis requires expertise in the Kconfig language as well as in-depth knowledge of Linux feature structures.

Developers and maintainers modifying Kconfig files can use our tool to assess the effect of the changes they perform on feature hierarchy. By querying `FMDiff` data, they can obtain the list of feature changes between their local version and the latest release. This will give them insight on the spread of a change by answering questions such as "*which features are impacted?*" and "*should this feature be impacted?*". Moreover, developers can follow the impact of changes performed by others on their subsystem, by looking at changes occurring on features of their sub-system.

As mentioned in Section 3, we do not track all possible changes occurring in Kconfig files. We ignored the `range` attribute, and only partially capture changes of `choice` structures. Those limitations were not problematic during the evaluation of `FMDiff` because the `range` attribute is not used widely (less than 170 occurrences in v3.10 kernel, for over 12,000 features) and in our small sample, choice modifications do not occur often. We consider that the loss of information is minimal. We will improve this in future work.

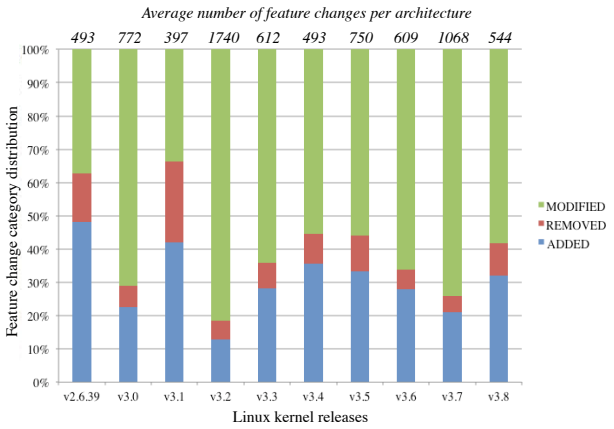

Figure 4: Evolution of the change category distribution (averaged over architectures)

6.2 Exploiting change types

Since the addition and removal of features in the Linux FM are well documented, we expect that features are also modified during the course of a release. We compute, over ten releases, the total number of changed features and the number of modified, added and removed features in each architecture specific FM; using only the first level of our change classification.

To obtain an overview of the changes occurring in each release, we compute the average number of modified, added and removed features per architectures. Based on `FMDiff` data, we found out that in release 3.0, the average number of feature changes in architecture specific FMs is 722. About 70% of those changes are modifications of existing features, 22% are additions of new features, and only about 8% of those changes are feature removals. The results are presented in Figure 4.

Overall, modifications of existing features account for more than 50% of the feature changes in most releases (8 out of 10). This means that feature modifications play a major role in the evolution of the Linux FM compared to the other changes (several features have been added while only few have been removed).

6.3 Threats to validity

Internal validity The evaluation of the tool was done by manually inspecting changes in Kconfig files and recorded changes in `FMDiff`. Like most manual processes, it is error prone. We recorded comparison and matches and we share the sample and results on our website for further validation.

The sampling of `FMDiff` changes for the validation is done randomly, so the different releases, architectures or change types are not equally represented in our samples. We consider that this sample contained enough different types of changes and feature operations to be representative of common feature transformations performed on the Linux FM.

External validity Our change classification, and tool are tightly linked to the Kconfig language. While a mapping between Kconfig and more generic FM (such as FODA) exists [14], we did not investigate its usage to generalize our approach. This work is currently limited to product lines using the Kconfig language as a mean to describe their features.

7. RELATED WORK

The Linux kernel has been used as an example of an evolving software product line many times in the past. Israeli et al. show in [5] that the Linux kernel evolution followed some of Lehman's Law of software evolution [6], namely the continuing growth by measuring the number of lines of code over time. Lotufo et al. [7] study the evolution of the Linux kernel variability model over time through FM structural metric evolution (model size, number of leaves, etc.). They show in their study that the number of features and constraints increase over time, but also that maintenance operations are performed to keep the complexity of the variability model in check. However, they do not provide details on change operations, nor ways to capture them in an automated way.

In order to study the Linux Kernel variability model structure, properties and evolution, several research teams have developed tools to reconstruct a FM from Kconfig files. LVAT [15] and `Undertaker` ([18],[20],[2]) are the main examples of such tools. We chose to rely on `Undertaker` for its convenient wrapping of `kdump`, allowing us to use the same tools that are also used by the Linux kernel development team. LVAT could have allowed us to capture the feature hierarchy. However, `kdump` flattening of the hierarchy facilitated the capture of feature hierarchy changes through changes of `depends` statements.

Several FM change classifications have been proposed in the past. In his thesis, Paskevicius describes several transformations that can be operated on a FM [9]. Similarly, FM change patterns have been identified by Alves et al. in [1] and Neves et al. in [8]. In his study of the co-evolution of models and feature mapping [13], Seidl also describes a set of operations applied to FM. Thuem et al. [22] classify feature changes based on their impact on the possible products that can be generated from the FM - a change can increase or decrease the number of products that can be obtained from a product line. More recently, Passos et al. ([10],[11]) compiled a catalogue of the evolution patterns occurring specifically on the Linux kernel.

We did not use those classifications in our study for two main reasons. First, according to She et al. [15] a `depends` statement can either be interpreted as a cross-tree constraint or a hierarchy relationship, so we cannot automatically map changes of `depends` statements in other change classifications. Second, FMDiff is able to capture changes in feature attributes which are not considered by these classifications.

8. CONCLUSION

In this paper, we presented a classification scheme to categorize changes in the Linux FM and the `FMDiff` tool to automatically extract these changes from two versions of Kconfig files declaring the Linux FM. We evaluated our approach by manually validating the changes extracted by FMDiff from ten releases of the Linux kernel. The results clearly show that our approach can capture feature changes accurately. Moreover, a comparison between the information on changes obtained with `FMDiff` and the information obtained through manual analysis of the textual differences between Kconfig files highlighted that our approach provides a more comprehensive view on FM changes. Finally, we demonstrated the usefulness of our approach and the information extracted on feature changes by providing a first glimpse on the evolution of the Linux FM over ten releases, finding that the majority

of changes refer to feature modifications.

As a next step, we plan to detail our studies on the evolution of the Linux FM by analyzing the fine-grained change types. Using the data acquired by `FMDiff`, we will answer questions such as what are the most frequent types of changes performed in the Linux FM and which features and parts of the feature model are changing frequently. Furthermore, since we rebuild architecture specific FMs, our dataset enables us to analyze and compare their evolution. Another direction of our future research is to investigate the impact of feature changes on other variability spaces, such as build and source code variability. For instance, we plan to explore how feature changes ripple through the Linux kernel implementation.

9. ACKNOWLEDGEMENTS

This publication was supported by the Dutch national program COMMIT and carried out as part of the Allegio project under the responsibility of the Embedded Systems Innovation group of TNO.

10. REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, page 201–210. ACM, 2006.

[2] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A robust approach for variability extraction from the linux build system. In *16th S.P.L.C. - Volume 1*, SPLC '12, page 21–30. ACM, 2012.

[3] H. Giese, A. Seibel, and T. Vogel. A model-driven configuration management system for advanced it service management. In *Proceedings of the 4th International Workshop on Models@run.time, Denver, Colorado, USA*, volume 509, pages 61–70. CEUR-WS.org, 10 2009.

[4] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. Consistency maintenance for evolving feature models. *Expert Systems with Applications*, 39(5):4987–4998, 2012.

[5] A. Israeli and D. G. Feitelson. The linux kernel as a case study in software evolution. *J. Syst. Softw.*, 83(3):485–501, 2010.

[6] M. Lehman. Laws of software evolution revisited. *Software process technology*, page 108–124, 1996.

[7] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the linux kernel variability model. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, number 6287 in Lecture Notes in Computer Science, pages 136–150. Springer Berlin Heidelberg, 2010.

[8] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the safe evolution of software product lines. *SIGPLAN Not.*, 47(3):33–42, 2011.

[9] P. Paskevicius, R. Damasevicius, and V. Štuikys. Change impact analysis of feature models. In T. Skersys, R. Butleris, and R. Butkiene, editors, *Information and Software Technologies*, number 319 in Communications in Computer and Information Science, pages 108–122. Springer Berlin Heidelberg, 2012.

[10] L. Passos, K. Czarnecki, and A. Wkasowski. Towards a catalog of variability evolution patterns: the linux kernel case. In *Proceedings of the 4th International Workshop on FOSD*, FOSD '12, page 62–69. ACM, 2012.

[11] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 91–100. ACM, 2013.

[12] D. Romano and M. Pinzger. Analyzing the evolution of web services using fine-grained changes. In *IEEE 19th International Conference on Web Services (ICWS)*, pages 392–399, 2012.

[13] C. Seidl, F. Heidenreich, and U. Aßmann. Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, page 76–85. ACM, 2012.

[14] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. *VaMoS*, 10:45–51, 2010.

[15] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 461 –470, 2011.

[16] 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 (SPLC-OSSPL 2007)*, page 30, 2007.

[17] J. Sincero and W. Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. *SPLC*, page 257–260, 2008.

[18] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. *SIGPLAN Not.*, 46(2):33–42, 2010.

[19] M. Svahnberg. *Variability in Evolving Software Product Lines*. PhD thesis, Research Board at Blekinge Institute of Technology, 2000.

[20] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, page 47–60. ACM, 2011.

[21] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or alive: Finding zombie features in the linux kernel. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, page 81–86, 2009.

[22] T. Thuem, D. Batory, and C. Kaestner. Reasoning about edits to feature models. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, page 254–264, 2009.