Delft University of Technology
Software Engineering Research Group
Technical Report Series

# Analysing Feature Model Changes using FMDiff

Nicolas Dintzner, Arie van Deursen, Martin Pinzger

Report TUD-SERG-2015-004

TU Delft

SERG

TUD-SERG-2015-004

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in *Software and Systems Modeling* journal, special issue on *Variability Modeling of Software Intensive Systems*

# Analyzing the Linux Kernel Feature Model Changes Using FMDiff

**Nicolas Dintzner · Arie van Deursen · Martin Pinzger**

**Abstract** Evolving a large scale, highly variable systems is a challenging task. For such a system, evolution operations often require to update consistently both their implementation and its feature model. In this context, the evolution of the feature model closely follows the evolution of the system. The purpose of this work is to show that fine-grained feature changes can be used to guide the evolution of the highly variable system.

In this paper, we present an approach to obtain fine-grained feature model changes with its supporting tool "FMDiff". Our approach is tailored for Kconfig-based variability models and proposes a feature change classification detailing changes in features, their attributes and attribute values. We apply our approach to the Linux kernel feature model, extracting feature changes occurring in sixteen official releases. In contrast to previous studies we found that feature modifications are responsible for most of the changes. Then, by taking advantage of the multi-platform aspect of the Linux kernel, we observe the effects of a feature change across the different architecture-specific feature models of the kernel. We found that between 10 and 50% of feature changes impact all the architecture-specific feature models, offering a new perspective on studies of the evolution of the Linux feature model and development practices of its developers.

Nicolas Dintzner, Arie van Deursen
Software Engineering Research Group,
Delft University of Technology, Delft, Netherlands
E-mail: {N.J.R.Dintzner}{Arie.vanDeursen}@tudelft.nl

Martin Pinzger
Software Engineering Research Group, University of Klagenfurt, Klagenfurt, Austria
E-mail: Martin.Pinzger@aau.at

**CR Subject Classification** D.2.9 Software engineering - Management - Software configuration management · D.2.13 Software engineering - Reusable Software - Domain engineering

**Keywords** software product line, feature model, evolution

## 1 Introduction

Software product lines are designed to maximize re-use of development artefacts while reducing development costs, through the identification and formalization of what is common and variable between different members of a product family [9]. Features, as configuration units, represent functionalities or characteristics that may be included in products of a product line. Available features are often formalized in a feature model, describing both the options themselves and their allowed combinations. The choice of features to offer to customers and their allowed configurations will influence every step of the development of the product line: its design, architecture, implementation techniques, and applicable methods to instantiate products from a set of assets (source code, scripts, resources) [9].

Over time, as a software product line evolves, features are added, removed, or modified and the associated assets should be updated accordingly. Software product lines are often long lived systems and the complexity of the system increases over time to the point where evolution operations become error-prone and specific approaches and tools become necessary [39, 42, 44]. We can find in the literature accounts of the issues arising during the evolution of such systems [1, 19, 42]. In a different domain, it has been shown that the analysis of fine-grained source code changes facilitates software maintenance [14]. Encouraged by such

results, we propose to explore a similar idea in the context of highly variable software: observing the details of the fine-grained evolution of a feature model to derive information about the evolution of the system.

Feature model evolution has been extensively studied in the past [15, 26, 41, 44]. These studies provide insights on which operations may occur on features, detailed examples of transformations occurring on large scale product lines - industrial and open source, and the evolution of feature model structural metrics (number of leaves, nodes, constraints). But it is interesting to note that studies detailing feature evolution scenarios, such as [21,25,30], tend to focus on transformation leading to (dis)appearance of complete features, not covering changes to existing features or constraints, leaving us with little knowledge about the details of such changes.

In this paper, we propose to elaborate and apply our existing tool supported approach to extract and classify fine-grained feature model changes in the Linux kernel feature model [12]. While the Linux kernel is not a software product line per se, it has the technical characteristics of such systems, among which an explicit variability model, which we assimilate to a feature model following the work by Sincero et al. [36,37], making this system an interesting case of highly variable software. We rely on our existing classification of feature changes, based on the Kconfig language.[1] We improved `FMDiff`, the supporting tool, to extract a larger corpus of data covering more than twenty architecture-specific feature models applied for over sixteen releases of the Linux kernel, from release 2.6.39 until release 3.14. We use the collected data to draw lessons about the evolution of the Linux kernel.

First, we are interested in discovering the frequent change operations affecting the feature model that developers perform over time. This data will allow us to see if the most commonly studied feature changes are also the most common change operations occurring on the features of Linux kernel. Several studies (*e.g.*, [17, 21, 27] ) quantified the addition and removal of features in the Linux kernel over time or present structural metrics of the kernel's feature model, such as the depth of feature structures or the number of leaf features in each release, but despite being often studied more detailed information can be obtained. This leads to our first research question: *RQ1: What are the most common operations performed on features in the Linux kernel feature model?* Over the studied time period, we found that the most common feature change operation on this system is also the one that is the least described by current research on variable system evolution, namely the *modification* of existing features (instead of merely adding or removing them).

Secondly, we know that the Linux kernel is designed to support many different processor architectures, each potentially differing widely from others in terms of supported features. In this study, we extract the Linux feature model on a *per-architecture* basis. While we study the evolution of all of those models, some studies restrict themselves to the study of one of them to extrapolate their findings on others [21]. We also note that developers working on the Linux feature model have, except in trivial cases, no means to know which architecture can be impacted by a feature change. We use `FMDiff` to compare the evolution of those different models, and answer the following research question: *RQ2: To what extent does a feature change affect all architecture-specific feature models of the Linux kernel?* Our data shows that the different architecture feature models follow very different evolution paths, and that between 10 and 50% of feature changes affect all architectures depending on the release. This suggests that extrapolation of observations done on the evolution of one architecture-specific feature model should be conducted with care, and points to a potential caveat in the Linux development process.

The key contribution of this paper is FMDiff, an approach to extract and automatically classify feature model changes from the versioning history of Kconfig-based feature models. Furthermore, the paper contributes 1) a feature model change classification scheme, focused on Kconfig-based variability models; 2) the FMDiff tool; 3) two studies with the Linux kernel feature model showing that changes to existing features constitute a large proportion of feature changes of the Linux feature model and showing that the evolution of architecutre-specific feature models of Linux follow different evolution path.

The remainder of this paper is organized as follows. Section 2 provides some background information on the Linux kernel, its feature model, and the tools we rely on to extract it. We present our feature change classification and its rationale in Section 3. `FMDiff` is introduced and evaluated in Section 4. We illustrate the capability of our tool in Section 5 by answering our two research questions. We reflect on the use of `FMDiff` and fine-grained feature changes in the context of the evolution of highly variable systems and product lines in Section 6. Section 7 presents related work. Finally, we conclude this paper and elaborate on potential future applications of `FMDiff` in Section 8.

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

```
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
```

**Listing 1** Example of a feature declaration in Kconfig

## 2 Background: The Linux kernel variability model

The approach described in this paper is based on the extraction of feature models (FMs) declared with the `Kconfig` language. In this section, we present general information regarding the `Kconfig` language, the Linux kernel that we used as a case study, and the model transformation we perform on the Linux feature model before analysis.

### 2.1 The Kconfig language

Kconfig is a variability modelling language used to describe configuration options (features) and their composition rules (cross-tree constraints). Listing 1 exemplifies the declaration of a configuration option in the `Kconfig` language.

In this work, we assimilate configuration options declared in the Kconfig language to features and the set of options with their constraints to a feature model [37]. The models created using Kconfig will differ from more standard feature models declared using FODA notation [18], but the constructs of both notations of can be mapped to one another [34].

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 loadable kernel 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 `if` keyword is satisfied. The text following the type on line 4 is the `prompt` attribute. It defines whether the feature is visible in the configuration tools 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 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. `depends`, `select` and constrained `default` statements are used to specify the cross-tree constraints of the Linux kernel FM. A feature can have any number of such statements.

Furthermore, `Kconfig` provides the means 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. Another possibility is to use `choices`. Such statement provides constructs similar to "alternative" (1 of) and "or" feature constraints (1 or more of) found in the FODA feature modelling notation [18]. A `choice` itself can also be subject to constraints and have dependencies expressed using `depends` statement.

Finally, features can have the "option" attribute, allowing the definition is a wide range of key/value pairs associated with features. This is used to flag features to be used in default (or generated) configurations for instance - option with the key "def_conf_list". Another usage is to tune the module resolution mechanism, or import additional variables.

`Kconfig` offers the possibility to define a 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. The configurator may also rely on the dependencies declared between features to create the displayed hierarchy. Constrains defined on menus and menuconfigs are applicable to all elements within. Menu can have the "visible" attribute, associated with a Boolean expression of features, complementing the "prompt" attribute. More details about the Kconfig language can be found in the official documentation.[2]

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

### 2.2 The Linux kernel

An example of system relying on the Kconfig language to manage its variability is the Linux kernel. Linux users can tailor their own kernel with `Menuconfig` (among other tools), the kernel configurator. This tool displays available configuration options in the form of a tree, and as the user selects or unselects options, the tree is updated to show only options that are compatible with the current selection.

Such tools use the textual descriptions of the Linux features contained with Kconfig files as an input, and provide a collection of selected features as an output, in the form of a list of feature names. During the configuration process, the configurator identifies the files to include and the features to display, depending on constraints expressed in those files. Constraints on file selection, or selectability of features, are resolved using naming convention based on feature names.

The choice of the target hardware architecture (*e.g.*, X86, ARM, SPARC) does not follow this rule. Because the choice of target architecture defines which file should be read first, it uses another mechanism. The name of the chosen architecture is defined during start-up (and can be modified later on) and stored in a variable used to build the first visualization of the FM ($SRCARCH, visible in "./Kconfig"). If no target architecture is given when starting the tool, it uses the architecture of the machine on which it is run by default. As a result, no parts of the Linux kernel FM represent the choice between architectures - while the architectures themselves are present as features.

This becomes important when rebuilding the Linux FM: without knowing which hardware architecture is being considered, we do not know which files to consider when rebuilding the FM. To avoid this problem, the methodology commonly applied is to rebuild a partial Linux FM per supported hardware architecture [21,23]. In this study, we use this specific approach when rebuilding the Linux FMs and analyzing FM changes.

### 2.3 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 tool, `Undertaker`, to translate Kconfig features into an easier to process format [43]. This tool has been used in the past for similar purposes. `Undertaker` uses it to reformat the Kconfig model before using it to determine feature presence conditions. It produces a set of ".rsf" files, containing annotated triplets formatted according to the "Rigi Standard Format" [40]. Each file contains an architecture-specific FM, *i.e.* an instance of the Linux FM where the choice of hardware architecture is predetermined.

```
1 Item ACPI_AC tristate
2 Prompt ACPI_AC 1
3 Default ACPI_AC "y" "X86 && ACPI"
4 ItemSelects ACPI_AC POWER_SUPPLY "X86 && ACPI"
5 Depends ACPI_AC "X86 && ACPI"
```

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

Listing 2 shows the example of the feature declared in Listing 1 in rsf triplets as output by `Undertaker`.

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.

`Undertaker` eases feature extraction but modifies their declaration. Among the applied modifications, two are most important for our approach: first, `Undertaker` flattens the feature hierarchy and then resolves features `depends` statements. Concerning the flattening of the hierarchy, `Undertaker` modifies the `depends` statement of each feature to mirror the effects of its hierarchy. For instance, `Undertaker` 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, `Undertaker` 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. Such transformations will influence the results of the comparison process and the interpretation of the captured changes. However, it has to be noted that the changes preserves the Kconfig semantics as described in [33].

## 3 Change classification

As mentioned in Section 2, the Linux feature model is expressed in Kconfig, describing both forward and backward dependencies with the "selects" and "depends" state-

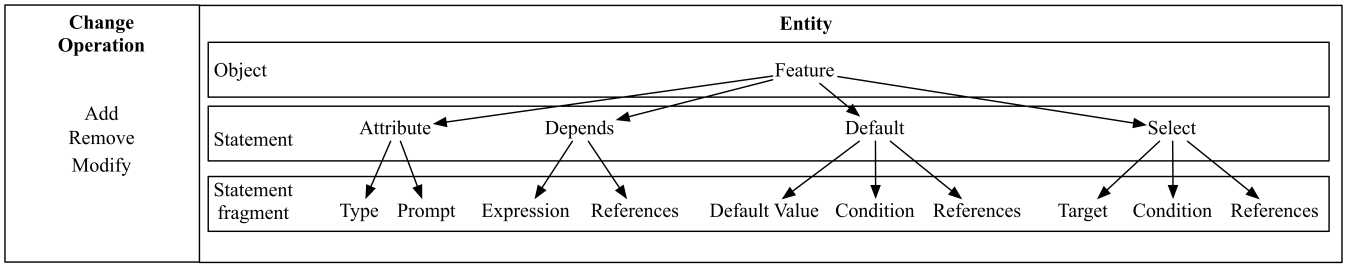


**Fig. 1** FMDiff 3-level feature model changes classification scheme.

ments. We aim at classifying feature changes occurring in the Linux kernel feature model (FM), capturing as accurately as possible the different changes that might occur on its statement. Existing feature change classifications [8,26] do not consider some specificities of the Kconfig grammar (*e.g.* select relationships with conditions). For this reason, we devise a new classification scheme, based on existing work, but specifically tailored for the Kconfig language.

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 on each level are named based on the modified entity (feature, statement and statement fragment), such as a `default` statement, and the change operation applied *i.e.* addition (`ADD`), removal (`REM`), or modification (`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 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 FMs expressed in the `Kconfig` language. Note that feature references contained in depend statements, select statements, and default value statements can only be added or removed as reference is either present or not. This leaves us with seven entities on which three operations are possible

and three for which we will consider only two - for a total of twenty seven change types.

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. `Undertaker` propagates feature dependencies of menus to the features a menu contains in the same way it propagates `if` block constraints. `Undertaker` 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, `Undertaker` 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, note 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 cannot 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`.

Currently, our change classification does not explicitly describe more complex feature model changes *e.g.* `merge feature` or `move feature`. Such changes can be viewed as a combination of simple changes described by our change classification. A merge operation would then result in the deletion of a feature and probably changes in the constraints of another one. The semantic of the change operation is lost (we cannot know it was a merge operation), but its effect on the FM itself is captured in the form of a set of change types.

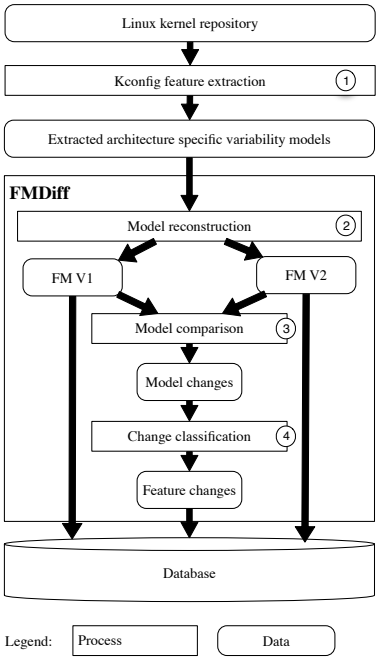


**Fig. 2** Change extraction process overview

## 4 FMDiff

In this section, we present our approach to automate feature change extraction and the tool that supports it: `FMDiff`. We then compare feature changes captured by `FMDiff` and changes observed in the original model. This allows us to evaluate the consistency of the changes captured with our approach and verify that `FMDiff` provides more information than textual differencing.

### 4.1 FMDiff Overview

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

#### *4.1.1 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 its source code repository.[3] Next, we use the `Undertaker` tool to extract architecture-specific FMs for each version. Undertaker outputs one ".rsf" file per architecture per version, in the format described in Section 2.

[3] Official Linux kernel Git repository: https://github.com/torvalds/linux

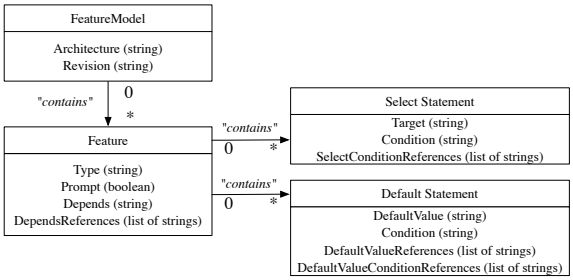


**Fig. 3** FMDiff feature metamodel

We perform a few noteworthy transformations when loading rsf-triplets into `FMDiff`. The rsf-triplets contain Kconfig choice structures, which are not always named in the Kconfig files. They are automatically renamed by `Undertaker` (*e.g.* `CHOICE_32`) guaranteeing the consistency of the rsf representation. Because the naming process is an automatic and does not depend on the content of choice, or its attributes, the same choice structure can be renamed differently in different versions. As a consequence, we cannot rely on naming to identify uniquely and reliably evolving choice structures. For those reasons, we ignore all choices when reconstructing the feature model from ".rsf" files. Note that the hierarchy constrains imposed by the choices are still reported on the relevant features during the hierarchy flattening process. However, we do lose information regarding mutually exclusive features.

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.

### *4.1.2 FMDiff feature model reconstruction*

As a second step, we reconstruct FMs from two consecutive 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 version of the FM. A `FeatureModel` contains any number of features represented as `Feature`. Each feature has a name, type (Boolean, tristate, 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 semantics associated with 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.

By using Undertaker and the rsf format as an input, we make a trade-off. The simple structure of the ".rsf" files facilitates the reconstruction of the Linux feature model. The hierarchy flattening give us, locally on each feature, additional information about constraints imposed by the hierarchy - allowing us to capture such changes later on. On the other hand, we cannot capture all feature attributes, we lose some information regarding choice structures - but preserve their induced constraints, and regrouping default value statements does not always respect Kconfig semantics. The consequences of this choice on the approach and the collected data are discussed in Section 6.

In the context of this study, we extended our dataset by including in it every rebuilt architecture-specific feature model. Once we obtain the .rsf representation of a Linux architecture specific model, we can proceed with the change identification and extraction.

### *4.1.3 Comparing models*

For the comparison of two FMs, `FMDiff` builds upon the the `EMF Compare`[4] framework. EMF Compare is part of the Eclipse Modelling Framework (EMF) and provides a customizable "diff" engine to compare models. It is used to compare models in various domains, like interface history extraction [31], or IT services modelling

[4] http://www.eclipse.org/emf/compare/

[13], and is flexible and efficient. EMF Compare takes as input a meta-model, in our case the meta model presented in 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 algorithm provided by EMF Compare is a two step process: first a matching phase, and then a diffing phase. The first step, the "matching" phase, identifies which objects are conceptually the same in the two instances. The diffing step uses items considered to be identical in two model instances to generate a list of model differences. Both steps need to be specialized for our study: we must provide matching rules, and a translation from EMF model changes to feature model changes.

To match features in two FMs, we rely on their name only: two features in two models represent the same concept if they have the same name. Note that this allows us to match features even if their dependencies or type have been modified. 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.

During the second phase, the "diffing" 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.

#### *4.1.4 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 analysed 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 version 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. We extract the information per architecture-specific FM. We build one database per architecture in which we store both the changes and the FMs.

### 4.2 Evaluating FMDiff

`FMDiff`'s value lies in its ability to accurately capture changes occurring on the Linux feature model (consistency) and its ability to provide information that would be otherwise difficult to obtain (interestingness). To evaluate `FMDiff` with respect to those two aspects, 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 consistent if it contains all changes seen in Kconfig files, and its data interesting if it provides more information than what can be obtained using textual differences. We start by describing the dataset used for the evaluation, and then assess them separately.

#### *4.2.1 Data set*

Using Git, we can navigate in the history of the Linux FM and extract snapshots that will be used for later comparison. It has been shown that the Linux FM is modified for corrective reasons during a release cycle [17, 21]. To avoid comparing feature model that might not be consistent with implementation, or simply do not reflect what was initially intended by the developer (a bug), we chose to compare only tagged releases. We noticed that few feature model changes were operated between the first release candidate version of a kernel and its last stable revision. For those reasons, we believe sufficient details can be obtained by extracting changes between stable official releases.

For all releases of the Linux Kernel from 2.6.28 to 3.14, we rebuild 26 architecture-specific FMs. We extract the changes occurring in 16 releases, over a time period of 3 years (from March 2011 for 2.6.38 to April

2014 for 3.14). This range of releases covers the first release supported by our infrastructure (Undertaker) up to the latest available release at the time of the study.

Between release 2.6.38 and 3.14, five 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 3.14. Our dataset contains 2,734,353 records describing the history of the Linux kernel FM.

#### *4.2.2 Consistency*

As mentioned in Section 4, the extraction and reconstructions of the Linux FM affects the data at our disposal during the comparison process, preventing us from obtaining certain types of changes (choices, range attributes,...). But, those exceptions aside, all other feature changes that can be observed in Kconfig files history should be also visible in `FMDiff` dataset. Changes not meeting this criteria would be signs of inconsistencies between the two representations of the same changes. To evaluate the consistency of the captured changes, we verify that a set of feature changes observed in Kconfig files are also recorded by `FMDiff`.

*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 change 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 consistent `FMDiff` data is. We also take into account that renamed features will be seen in `FMDiff` as "added" and "removed".

*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 `Undertaker` 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 described this 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 consistent with respect to the changes occurring on the Linux FM.

#### *4.2.3 Interestingness*

Developers and maintainers of the Linux kernel often work on features. Changes on features might affect the ones they work on, or their direct dependencies. To identify such changes, textual differencing tools in combination with repository history navigation facilities can be used (such as GitK for Git repositories). Inspired by the work of Ying et al. [46], we propose here to compare the information that can be obtained by textual differences and using `FMDiff` to evaluate the interestingness of the collected data. We will consider that `FMDiff` provides "interesting" information for developers and maintainers if it makes available information otherwise difficult to obtain.

*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 `Undertaker` propagating dependency changes onto other feature attributes or onto its sub-features (*e.g.* when a `depends` statement is modified on a parent feature). Here, *indirect matches* indicate that `FMDiff` captures side-effects of changes made on Kconfig files, more difficult to observe using textual differences.

*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, resulting in 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 that could not be explained; but the change was consistent with the content of `Undertaker`'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, those 79 indirect matches indicate that `FMDiff` data contain 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.

## 5 Using FMDiff to Understand Feature Changes in the Linux Kernel Feature Model

`FMDiff` captures changes occurring on features of the Linux kernel and stores each individual change in a database. Thanks to this format, we can easily query the gathered information to study the evolution of the kernel feature model (FM) over time. We use this information to identify the most common change operations performed on features and study the pervasiveness of feature changes across the multiple architecture-specific FMs of the kernel, and to answer the research questions as raised in the introduction.

```
1 select count(distinct feature_name)
2   from fine_grain_changes
3   where revision='v3.0'
4   and change_category='MOD_FEATURE'
```

**Listing 3** Example of query on FMDiff data: modified features in release 3.0

### 5.1 High-level view of the Linux FM evolution

FMs, as central elements of the design and maintenance of SPLs, have attracted substantial attention over the past few years in the research community. For example, several studies describe practical SPL evolution scenarios related to FM changes [25, 30, 32], focusing mostly on addition and removal of features. An open question, however, is whether the changes commonly studied are also the most frequent ones on large scale systems. This leads us to our first research question, which we answer using `FMDiff` data. *RQ1: What are the most common operations performed on features in the Linux kernel feature model?*

Let us consider the highest level of changes that `FMDiff` captures: addition, removal and modification of features. We use our database to query, for a given architecture, features that were changed during a specific release. Listing 3 shows an example of such query, giving us the number of features modified during release 3.0 for a single architecture. We compute, for sixteen 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 average number of modified, added, and removed features per architecture.

As shown in Figure 4, during release 3.0, the average number of feature changes in architecture-specific FMs were 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. Note that the total number of architectures taken into account varies over time. In Figure 4,

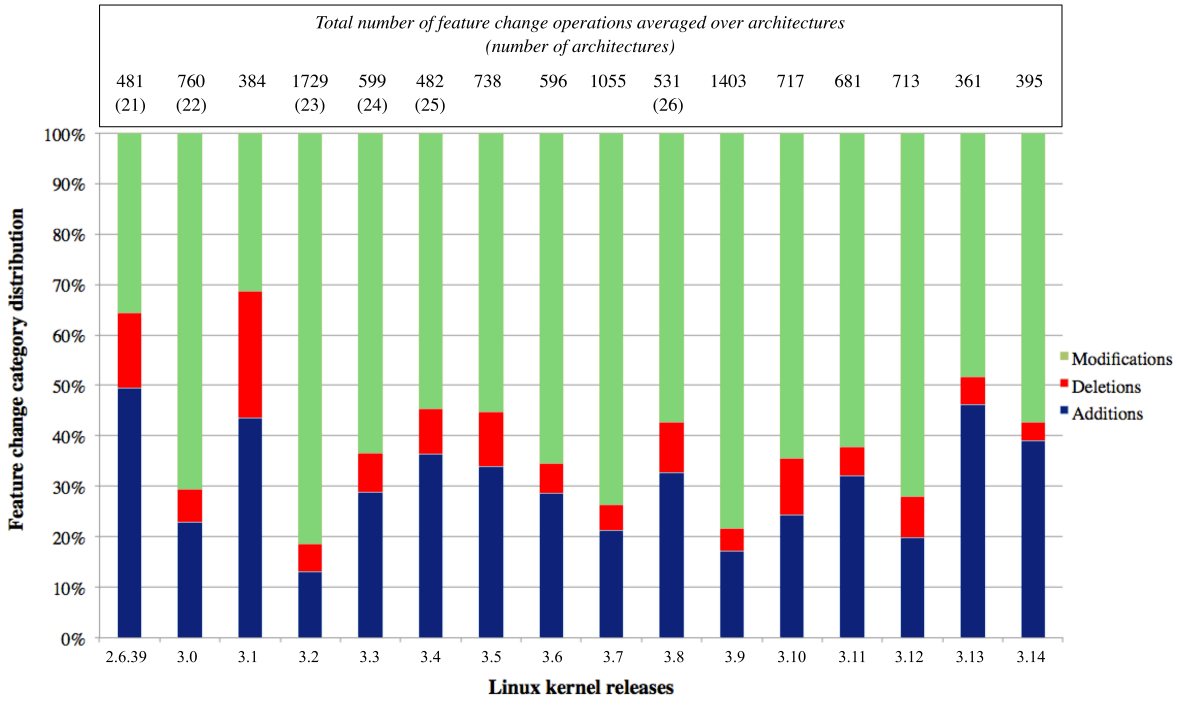


**Fig. 4** Evolution of the feature change category distribution (averaged over architectures)

the number of architectures used for the computation of the graph is noted in parenthesis above each column.

Over the 14 studied releases, on average per architecture, creation of new features accounts for 10 to 50% of feature changes. Deletion of features accounts for 5 to 20% of all feature changes and modification of existing features accounts for 30 to 80% of all feature changes.

In this case, modifications of existing features include modification of their "depend statement". Such statements are affected by direct developer action (edition of the feature attribute in a Kconfig file), or by changes in the feature hierarchy, as the hierarchy is used during FM extraction (see Section 2).

With this information, we can answer our first research question. Modifications of existing features account, on average, for more than 50% of the feature changes in most releases (13 out of 16), making them the most frequent high-level feature change occurring on the Linux kernel FM. This clearly shows that modifications of existing features is a common operation during the evolution of the Linux FM compared to the other changes (adding and removing features). This conclusion above is specific to certain types of representations of FMs. In the most common FODA notation, cross-tree constraints refer to features, but are attached to a FM rather than to the features themselves. A modification to a cross-tree constraint is arguably different than a feature modification. In this specific case, because cross-tree constraints are part of the definition of a given, well-specified feature, we can make such claim.

## 5.2 Evolution of architecture-specific FMs

In this section, we compare the evolution of the different architecture-specific FMs. Our aim is to assess how similar their evolution is and answer our second research question: *RQ2: To what extent does a feature change affect all architecture-specific FMs of the kernel?*

### *5.2.1 Motivation*

The Linux kernel feature model (FM) has been extensively studied as an example of highly variable system. In order to analyse the evolution of its FM, a common assumption is that all hardware architecture-specific FMs supported by the kernel evolve in a similar fashion [21]. This implies that observations made on a single architecture can be, and are, extrapolated to the entire kernel. Such approaches are justified by the fact that the different architectures share up to 60% of their features [11], and that the growth rate of architecture-specific FMs are similar [21]. By comparing the evolution of the different architecture-specific FMs, we see under which condition such extrapolations hold.

We propose here to observe the evolution of those feature models in regard to the development practices applied by developers. The Kconfig file structure makes a clear distinction between features that are meant to be used for a single architecture (organized in a subfolder of the main "arch" directory), and the others. This provides guidance to developers during maintenance, about where to declare those very specific features. However, every subsystem of the kernel (memory, file system, drivers,...) can contain architecture specific features.

In practice, when a change is applied to a configuration option in a Kconfig file, there is no guarantee that this change is affecting all architecture-specific FMs in a similar way. Concrete examples of such changes can be found by browsing through the Linux kernel source code repository history. During release 3.0, feature ACPI_POWER_METER was removed and replaced by SENSORS_ACPI_POWER contained in another code module.[5] We can observe that the ACPI_POWER_METER feature is removed from the file "/drivers/acpi/Kconfig" file, and that SENSORS_ACPI_POWER is added to "/drivers/hwmon/Kconfig". The same change is captured by `FMDiff` in the form of the removal of ACPI_POWER_METER and the addition of SENSORS_ACPI_POWER. Using our database, we can observe that the removal of the ACPI_POWER_METER only affected two architectures: x86 and IA64. However, the addition of SENSORS_ACPI_POWER can be seen in x86, IA64, and ARM. Given the commit message, it is unclear whether this was the expected outcome or not. The change does not seem to have been reverted since then.

Another example is the addition of an existing feature to an existing architecture-specific FM. Also in release 3.0, feature X86_E_POWERSAVER pre-existing in the X86 architecture was added to other architectures and its attribute modified. By searching the Git history, we identified the commit[6] removing this feature from "arch/x86/kernel/cpu/cpufreq/Kconfig" and moving it to "drivers/cpufreq/Kconfig.x86" with a modification to "drivers/cpufreq/Kconfig" to include the new file, with a guard statement checking the selection of the X86 feature. Using `FMDiff` data, we can observe that in release 3.0, the depend statement and select condition attributes of these features were modified in X86 (adding references to the X86 feature) in the X86 FM as a result of a change in the feature's hierarchy. However, it is, for instance, also seen as added in ARM and other architecture-specific FMs.

Such changes can be problematic as a thorough testing practice would require validating a change for all architectures. The first level of verifications that developers can use is simply to compile a specific configuration. Errors in the Linux feature model often result in errors during compiling certain configurations [1]. When a developer modifies the behaviour or capabilities of the kernel for multiple architectures, he needs to "cross-compile" their modifications and ensure that the modifications behave appropriately on all of them. This is also true when a modification of the FM affects an architecture-specific feature, or if an architecture-specific change is applied to a feature. However, the cross-compilation process is non-trivial.[7]

Even with a specific tool chain, it appears that cross-compilation is inconsistently done during the development process as reported by the Linux development team in commit messages, such as

> "Untested as I don't have a cross-compiler." [8]

> "We have only tested these patchset on x86 platforms, and have done basic compilation tests using cross-compilers from ftp.kernel.org. That means some code may not pass compilation on some architectures." [9]

or this message posted by Linus Torvalds in the Linux kernel mailing list

> "I didn't compile-test any of it, I don't do the cross-compile thing, and maybe I missed something." [10]

We find ourselves in a situation in which, following a feature modification, identifying the impact across architectures is non-trivial, and cross-compilation, the first mean to validate such changes, is not applied consistently. There are many developers working on the kernel, and a few not cross-compiling might not affect the quality of the end-product. However, if we consider a practical evolution scenario, a change will affect only certain combinations of features. If a developer does not cross-compile, then, others will have to know which configurations were affected in order to validate them on different platforms. Considering the number of configurations of the kernel, we can wonder how likely it is for others to test the appropriate configurations. But if such cross-architecture feature changes are rare, such practices would be reasonably safe.

[5] commit: 7d0333
[6] commit: bb0a56
[7] Linux cross-compilation manual: http://landley.net/writing/docs/cross-compiling.html
[8] commit: 2ee91e
[9] commit: cfa11e
[10] https://lkml.org/lkml/2011/7/26/490

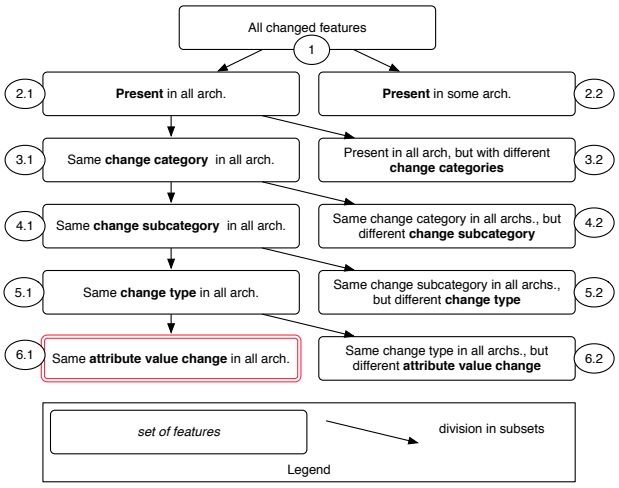


**Fig. 5** Extracting feature changes affecting all architectures

The comparison of the evolution of the different architecture-specific feature models of the Linux kernel allows us to assess the validity of extrapolations of observations based on feature changes of one architecture to others, and reflect on the development practices mentioned above.

### *5.2.2 Methodology*

To analyse the discrepancy between the evolution of the different architecture-specific FMs, we compare the changes occurring on the features of the different FMs during the same release. We proceed as shown in Figure 5.

We first identify which features were changed in all architectures for a given release. This is achieved by querying all changes of all architecture-specific FMs for a given release from the `FMDiff` database. Then, we isolate unique feature names from that set. We obtain a first list of feature names (marked as "1" in Figure 5). We split that set into two: features that are seen as changed in `FMDiff` data in all architecture-specific FMs, and those that are seen changed in only some architectures. This gives us the feature sets marked as "2.1" and "2.2" in Figure 5.

Using the set of features that appear in all architecture-specific FM changes, we compare the change categories associated with those features. This way, we check whether the main change operation (add/remove/modify) is the same on that feature in all architecture-specific FMs. Once again, we split the initial set of features in two: those that have the same *change category* in all architectures (set "3.1") and those that have different change categories (set "3.2").

We continue in a similar fashion by comparing the *change category*, *sub category*, *change type* and attribute change, always starting with the set of feature changes common to all architectures. Ultimately, we obtain the number of features that are seen as changed exactly in the same way in all architectures (set "6.1" in Figure 5). We repeat those steps for all available releases in the `FMDiff` dataset.

The comparison process is different when comparing feature changes based on attribute value changes, as this comparison is not sensible for all attributes. Because of the flattening of the Linux feature hierarchy, the same feature can have different attribute values (depend statements for instance) in different architecture-specific FMs. If a change is performed on such a statement, checking if the old and new values of a feature attribute are the same in different architectures will yield negative results: the value is different to start with, so even if the same change is applied, attribute values remain different.

This applies to all attributes consisting of Boolean expression of features: depend statements, select and default value conditions: 9 out of the 27 *change types* we identified in Section 3. Those attributes are ignored during the construction of the last sets ("6.1" and "6.2"). Because we capture changes in feature references on those attributes, we can still identify if a change affected such attributes in a similar fashion in all architectures. In fact, comparing these attribute changes would require to perform a semantic differencing on those attributes, rather than the textual comparison we do at the moment. We defer this to future work.

### *5.2.3 Experimental setup*

To answer our second research question using the methodology just described, we consider the following architecture-specific FMs: alpha, arm, arm64, avr32, blackfin, c6x, cris, frv, hexagon, ia64, m32r, m68k, microblaze, mips, mn10300, openrisc, parisc, powerpc, s390, score, sh, sparc, tile, unicore32, xtensa, and finally, x86. We remove from the set of considered changes, all changes caused by the introduction of a new architecture. For instance, when the architecture C6X is introduced in release 3.3, we observe in our dataset the creation of this FM and the creation of all of its features. During our comparison, all features will be seen as added in the C6X architecture-specific FM, introducing a large number of architecture-specific changes while in reality, the features have not been touched. To avoid this, we only include an architecture-specific FM one release after its initial introduction.

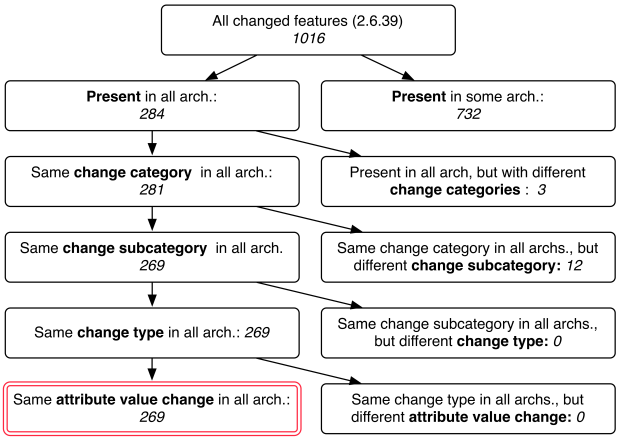


**Fig. 6** Example of architecture evolution comparison for release 2.6.39

For analysis purposes, we isolate the intermediate results so that features that evolved differently in different architectures can be isolated and the differences later manually reviewed. The analysis is performed using R scripts, directly querying the `FMDiff` database. The scripts are available in our code repository.[11]

### *5.2.4 Results*

By applying the methodology described in Section 5.2.2 for a single release, we obtain the information depicted in Figure 6. We can read this figure as follows: in release 2.6.39, 1016 features were changed. Out of those, 284 are seen as changed in all architectures (generic), while 732 are seen as changed in only some of them (architecture-specific). 281 of the features changed in all architectures have the same *change category*. 3 of them have different change categories in different architectures. This occurs when a feature is seen as added in an architecture-specific FM and modified in others for instance. 269 features have the same *change category* and *change subcategory* in all architecture-specific FMs, 12 do not. This occurs when features with different attributes in different FMs are deleted for instance. All those 269 changed features have the same *change type* and their attributes are changed in the same way in all architectures. Finally, we can see that out of 1016 changed features, only 269 changed in the exact same way in all architecture-specific FMs.

We apply the same methodology for all 16 official releases of the Linux kernel, and compile the results in Table 1. In this table, each release column is read like the diagram depicted on Figure 6, presenting the number of changed features affecting all (generic) or some (arch-specific) architecture-specific FMs, decomposed by change operation granularity - touched, change category, sub-category, types and down to attribute value. From this table, we learn the following.

First, the total number of changed features in each release, shown in the second row of Table 1, is very variable. Over the studied period of time, the release with the smallest amount of changed features is 3.1, with only 567 changed features, and the release with the largest number of changed features is release 3.11, with 4556. If we consider that the Linux kernel feature model contains approximately 12,000 features; in each release between 4 and 38% of the total number of features is touched.

Secondly, the difference between the evolution of architecture-specific FMs lies in the features being changed, not in the nature of the change applied. We can see in Table 1 that for each release, the largest difference between the number of generic and architecture-specific feature changes is found at the highest comparison level: a feature is touched in all architectures if it is seen as added, removed, or modified in all architectures - regardless of the exact change type (as shown in the third row of Table 1).

Finally, no features have architecture-specific change type and attribute value changes. In all releases, the number of architecture-specific change types and attribute value changes is zero. If a feature saw its statements changed in the exact same way in all architectures, then, according to our dataset, the details of those changes will be the same in all architectures as well (change type and attribute value).

As mentioned in Section 5.2.2, we do not isolate changes made to all attributes. This causes small discrepancies in the values shown in Table 1. For instance in release 3.4, we can see 257 features that have the same *change type* in all architectures but 252 with the same attribute changes in all architectures and 0 with different attribute changes. In this release, five features saw their attributes modified in slightly different ways in different architectures, however none of those attributes are tracked - relating only to Boolean expression of features. Such features are removed from the dataset before the comparison of attribute values, hence the potential drop in the number of features during this step.

The number of observed changed features in release 3.11 is surprisingly high compared to other releases. The architecture that changed the most during this release is the CRIS (Code Reduced Instruction Set) architecture. By manually inspecting the changes using Git and our dataset, we found a commit[12] modifying

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

[12] commit: acf836

| **Release** | **2.6.39** | | **3.0** | | **3.1** | | **3.2** | |
|---|---|---|---|---|---|---|---|---|
| Number of changed features | 1016 | | 1020 | | 567 | | 2361 | |
| | generic | arch-specific | generic | arch-specific | generic | arch-specific | generic | arch-specific |
| Touched | 284 | 732 | 600 | 420 | 213 | 354 | 931 | 1430 |
| Change category | 281 | 3 | 600 | 0 | 212 | 1 | 922 | 9 |
| Sub category | 269 | 12 | 596 | 4 | 202 | 10 | 921 | 1 |
| Change type | 269 | 0 | 596 | 0 | 202 | 0 | 921 | 0 |
| Attr. value | 269 | 0 | 596 | 0 | 202 | 0 | 921 | 0 |

| **Release** | **3.3** | | **3.4** | | **3.5** | | **3.6** | |
|---|---|---|---|---|---|---|---|---|
| Number of changed features | 946 | | 778 | | 1103 | | 823 | |
| | generic | arch-specific | generic | arch-specific | generic | arch-specific | generic | arch-specific |
| Touched | 232 | 714 | 274 | 504 | 455 | 648 | 298 | 525 |
| Change category | 231 | 1 | 265 | 9 | 435 | 20 | 287 | 11 |
| Sub category | 228 | 3 | 257 | 8 | 434 | 1 | 285 | 2 |
| Change type | 228 | 0 | 257 | 0 | 434 | 0 | 285 | 0 |
| Attr. value | 228 | 0 | 252 | 0 | 432 | 0 | 281 | 0 |

| **Release** | **3.7** | | **3.8** | | **3.9** | | **3.10** | |
|---|---|---|---|---|---|---|---|---|
| Number of changed features | 1385 | | 963 | | 1773 | | 1299 | |
| | generic | arch-specific | generic | arch-specific | generic | arch-specific | generic | arch-specific |
| Touched | 415 | 970 | 299 | 664 | 1042 | 731 | 430 | 869 |
| Change category | 412 | 3 | 292 | 7 | 1034 | 8 | 428 | 2 |
| Sub category | 406 | 6 | 284 | 8 | 1029 | 5 | 420 | 8 |
| Change type | 406 | 0 | 284 | 0 | 1029 | 0 | 420 | 0 |
| Attr. value | 403 | 0 | 283 | 0 | 1024 | 0 | 417 | 0 |

| **Release** | **3.11** | | **3.12** | | **3.13** | | **3.14** | |
|---|---|---|---|---|---|---|---|---|
| Number of changed features | 4556 | | 1406 | | 620 | | 704 | |
| | generic | arch-specific | generic | arch-specific | generic | arch-specific | generic | arch-specific |
| Touched | 615 | 3941 | 678 | 728 | 329 | 291 | 379 | 325 |
| Change category | 380 | 235 | 678 | 0 | 329 | 0 | 378 | 1 |
| Sub category | 375 | 5 | 678 | 0 | 328 | 1 | 378 | 0 |
| Change type | 375 | 0 | 678 | 0 | 328 | 0 | 378 | 0 |
| Attr. value | 370 | 0 | 674 | 0 | 326 | 0 | 374 | 0 |

**Table 1** Quantitative comparison of generic and "architecture-specific" feature changes

```
(...)
-source "drivers/char/Kconfig"
+source "drivers/Kconfig"

 source "fs/Kconfig"

-source "drivers/usb/Kconfig"
(...)
```

**Listing 4** Extract of the diff of file "/arch/cris/Kconfig" in release 3.11

the CRIS architecture configuration file (/arch/cris/Kconfig). The modification, shown in Listing 4, removed the inclusion of a specific set of drivers and replaced it by the inclusion of all standard drivers. This is a major contributor to the number of added features in the CRIS architecture-specific FM.

Finally, we consolidate our results in Table 2. For each release, we present the total number of changed features and the percentage of those features that are seen as changed exactly in the same way in all architecture-

| Linux Kernel release | Total number of changed features | % of changed features affecting all architectures |
|---|---|---|
| 2.6.39 | 1016 | 26.47 |
| 3.0 | 1020 | 58.43 |
| 3.1 | 567 | 35.62 |
| 3.2 | 2361 | 39.00 |
| 3.3 | 946 | 24.10 |
| 3.4 | 778 | 32.39 |
| 3.5 | 1103 | 39.16 |
| 3.6 | 823 | 34.14 |
| 3.7 | 1285 | 29.09 |
| 3.8 | 963 | 29.38 |
| 3.9 | 1773 | 57.75 |
| 3.10 | 1299 | 32.10 |
| 3.11 | 4556 | 8.12 |
| 3.12 | 1406 | 47.93 |
| 3.13 | 620 | 52.58 |
| 3.14 | 704 | 53.12 |

**Table 2** Evolution of the ratio of feature changes impacting consistently all architectures supported by the Linux kernel

specific FMs. We can read Table 2 as follows: in release 3.12, 47.93% of the 1406 changed features were seen as changed consistently in all architecture-specific FMs of the Linux kernel.

#### *5.2.5 Architecture-specific evolution*

With the gathered data, we can answer our second research question. *RQ2: To what extent does a feature change affect all architecture-specific FMs of the kernel?*

The data shown in Table 2 highlight that for a specific feature change in a release, it is very likely that this feature change affects only certain architecture-specific FMs. In that sense, observations related to FM evolution obtained by the study of a single architecture-specific FM cannot be generalized to all architectures, or help draw conclusions on the evolution of the overall Linux FM. Table 1 emphasizes that most feature changes might not even be seen in other architectures. It is interesting to note that, during release 3.11, while 4556 features were changed during the release, the average number of changed features per architecture is 681 (see Figure 4). This further supports our assumption that architecture-specific FMs evolve differently.

Table 1 also shows that if a feature is seen as changed in all architectures, in a large majority of cases, the change applied to the feature is the same. A good example of this is release 3.12, where among the 678 changed features that affected all architectures, all had the same *change category*, *change subcategory*, *change type* and attribute changes. In other cases, when there are discrepancies between how a changed feature affects different architectures, the discrepancy is in the *change category*: a feature is seen as modified in one architecture and added to another. In release 3.11 where 615 changed features affected all architectures, 235 had inconsistent change categories across architecture-specific FMs. This matches our observation regarding the addition of many drivers to the CRIS architecture FM in Section 5.2.4.

To conclude and answer RQ2, we can say that relatively few feature changes affect all architecture-specific FMs of the Linux kernel. We also note that a large majority of changes affecting all architecture-specific FMs affect them in the exact same way.

## 6 Discussion

The main objective of this paper is to support the maintenance and evolution of large scale software product lines (SPLs). We first reflect on the capabilities of FMDiff, the nature of the captured information, the results of our data analysis. Then, we continue by discussing the threats to validity of this study.

### 6.1 Fine-grained feature changes

Thanks to `Undertaker` 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. As mentioned in Section 4.2, `FMDiff` captures accurately a large majority of feature changes applied to the Linux kernel FM. Using FMDiff, feature changes are stored as lists of statement changes with the attribute values before and after the change (following our classification). Developers and maintainers modifying Kconfig files can use our tool to assess the effect of the changes they perform on the 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.

The extraction of fine-grained feature changes allowed to show that modification of existing features was a very frequent change occurring on the Linux feature model. If we look at previous research on the evolution of highly variable systems [17, 21, 25, 27, 30], we can see that the focus is put mostly on scenarios leading to the apparition or removal of features (such as add, remove, merge, or split). In the context of Linux, extending those studies to cover the modification of existing features would be beneficial. The data collected by `FMDiff` will help in such endeavours, pinpointing instances of such scenarios in this history of Linux kernel FM.

### 6.2 Architecture specific evolution

The comparison of architecture-specific FMs evolution showed us that those FMs evolved differently. The proportion of feature changes affecting all architectures varies between releases from 10 to more than 50%. We also see that, if a change affects all architectures, in almost every cases, the change is the same in all architectures. This limits the validity of extrapolating ob-

servations about FM evolution from one architecture to others. However, it is interesting to note that, once we determine that a change is visible in all architectures, we can safely assume that the modification is the same. Future studies of the Linux kernel feature model evolution using a similar feature model reconstruction technique should be clear about the studied architectures, as this will influence the results.

For this study, we focused on feature changes that affected exactly all architectures. An alternative would have been to identify clusters of architectures evolving more similarly than others. For instance, we can imagine that the evolution of the ARM has more in common with the ARM64 architecture than the X86. Then, it would be possible to extrapolate observations, not to all, but to a well defined set of architecture-specific FMs. The data collected during this study could be of use to identify such clusters.

The amount of changes affecting all architectures puts us at odd with respect to the development practices of the Linux developers. On the one hand, our data shows that feature changes visible in all architectures occur in every release, in large proportion. On the other hand, in Section 5.2, we show anecdotal evidence that developers are not inclined to cross-compile. We can assume that the delivered assets compile - at least for the architecture on which the developer was working. With more than 13,000 features, the number of possible configurations of the kernel is immense. Given that modifications to features will only affect specific configurations, only the developers and experts will know which configurations should be tested. So the changes might remain untested and a faulty feature could be delivered. Then, if this happens, the criticality of such problems will depend on how frequently this feature is used on the various platforms. We have to keep in mind that as long as the feature is not mandatory for a system, the problem can simply be fixed by not including it in the configured kernel image. Perhaps such errors are not critical nor frequent enough to warrant the use of much heavier testing practices.

Nonetheless, as shown by our data, cross-architecture feature changes occur frequently. In such situations, developers do not seem to have the means to identify which architectures might be affected by their changes, and do not consistently test. A tool, such as `FMDiff`, can capture the impact of feature changes across architectures. With this additional information, developers would have a better view of how often their modifications affect different architectures, making them more aware to such situations. If they wish to cross-compile their code, then `FMDiff` would give them a list of the impacted architectures to consider first.

### 6.3 Threats to Validity

*Construct validity* We first discuss the methods we used to extract changes from the Linux kernel feature model and their impact on the usage of the resulting data to reflect on the evolution of the Linux kernel FM.

A threat to the validity of our study is the representativeness of changes observed on a transformed version of the Linux FM when reasoning about its evolution. After extracting the Linux FM using Undertaker, the hierarchy is flattened and the constraints propagated on feature attributes. As a consequence, the changes captured by `FMDiff` include the edits performed by developers on Kconfig files as well as their consequences on the other features of the model. After the model transformation, we cannot differentiate between developer edits in the Kconfig files (human operation) and the propagated effect of those changes on other features. Following this, we transform the Undertaker model into an EMF model for comparison purposes, further modifying the data we use for this study. We argue that both developer edits and their propagated effects are relevant for the study of the evolution of the Linux FM. The transformation performed by Undertaker adheres to the Kconfig semantics as described in [33] (except for the "range" attribute, which is not extracted). This comforts us in the idea that the transformed model in the ".rsf" format produced by Undertaker can be used as a mean to study the evolution of the Linux FM. The model transformation from ".rsf" to EMF does not preserve the semantics of the Kconfig language, as we do not keep track of the order of certain attributes (such as default statements), and we do not consider `CHOICE` elements. Our dataset cannot be used to reflect on the evolution of the allowed configurations of the Linux kernel: we cannot tell which configurations were added or removed by looking at the feature changes captured by `FMDiff`. But, as we have shown in Section 4.2, the changes captured by `FMDiff` are consistent with the changes observed in Kconfig files. For those reasons, we are confident that the gathered data can be used to observe and reflect on feature changes occurring in Kconfig models.

Over time, the Kconfig language has evolved, and modifications to the constructs of the language should influence our change classification and comparison process. We did not take this into account for this study and we might miss new attributes or attempt to capture information no longer relevant. This constitutes our second threat to validity. To mitigate the effects of potential language evolution, we restricted the scope of releases we studied. Release 2.6.38, the first of our study, is the oldest release for which our version of Under-

taker was able to extract all architecture-specific FMs. We extended the scope of releases from there up to the most recent release at the time of writing (3.14). Using Git, we manually inspected the history of the Kconfig parser and grammar in the Linux repository (the "./scripts/kconfig" folder). We found a minor modification to value attribute (long integer allowed on value based features for instance).[13] We also found modifications to the allowed values on feature attribute "option" [14] as mentioned in Section 2, irrelevant in the context of this study. The other changes occurring during the studied releases were, as far as we can see, modifications to Kconfig internals, with no impact on the information captured by `FMDiff`. We have to consider that for a study over a longer period of time, we would have to take into account those changes, adapt the tool and classification in accordance to the evolution of the language.

As reported in Section 3 and mentioned here, information is lost during the model transformation and comparison process. The third threat to validity we consider is the influence of the missing information on our validity of the resulting dataset. The "range" feature attribute is not extracted and as such not used during comparison. `CHOICE` structures, present but with a specific naming convention, are removed from our intermediate model. However, the `range` attribute is not used widely (less than 170 occurrences in 3.10 kernel, for over 12,000 features) and for this reason we do not believe that this influenced our results or conclusions. During our manual evaluation of FMDiff, we found no occurrence of changes on `CHOICE` structures, comforting us in the idea that this is not a common change. But we assume that such changes can occur and would be overlooked by `FMDiff`. Changes to `CHOICE` structure would impact the contained features - the hierarchy flattening transformation ensures this. While we do not capture `CHOICE` changes, we can still observe their effects on features. For those reasons, we believe the loss of information has a minimal impact on our observations but must be taken into account for further analysis.

*Internal validity* With those limitations in mind, we reflect on the limits of our conclusions on the evolution of the Linux FM.

A threat to the internal validity of our study is the effect of the hierarchy flattening transformation on the number of observed feature modifications. When a `Menu`, `Menuconfig`, `Choice`, or `If` construct is modified by developers, changes to its dependencies will be reflected on the features it contains. As direct consequence, we will observe more feature modifications than if we looked at the actual edits performed by the developers, increasing the number of observed modifications of existing features. We would argue here first that the modifications do occur: the features are indeed modified, but indirectly. In that sense, the captured information is accurate, and does reflect the actual state of features in the feature model. Considering the overwhelming majority of modification of existing features in certain releases (more than 70% in release 3.7), we believe that our conclusion holds: feature modifications are, if not the most, at least a very common type of change on every observed release.

Concerning the comparison of architecture-specific FMs, we can question the model reconstruction process. The fact that a feature is included in an architecture-specific FM does not necessarily mean that the feature is selectable (dead feature). We might observe cross-architecture feature changes, that, in practice, do not affect the possible configurations of architecture-specific kernel images. As we do not take this into account, this constitutes the second threat to the internal validity of this study: a number of cross-architecture feature changes we observe in our dataset do not affect the allowed configurations described by those FMs. As mentioned as a threat to construct validity, our change extraction cannot capture semantic changes occurring on Kconfig-based systems. For this study, we restrict ourselves to capturing syntactic changes on features and offering a different view of those changes, leaving the semantic interpretation of the changes to experts. We consider in this study that a change to a non-selectable (but present and declared) feature could actually lead to making it selectable and should be reported and accounted for as a cross-architectural change, despite their potential lack of effects on the configurations of the architecture specific FM. For this reason, the absence of distinction between selectable and non-selectable features in our approach does not influence our conclusion. However, this further supports the fact that `FMDiff` data should not be used to reflect on the possible configurations of the system, but only on feature changes.

*External validity* We now reflect on the generalizability of our approach and its applicability in different contexts.

The first threat to the external validity of this approach is the use of a specific Kconfig-centered change classification. Our feature change classification is tightly linked to the Kconfig language, and would be difficult to re-use in other contexts. However, Kconfig is used in a number of highly variable systems [6], all of which could re-use directly our change classification.

[13] commit: 129784ab
[14] commit: 6902dccfda

The second threat to the external validity is the lack of application of `FMDiff` on other systems than Linux. The implementation of `FMDiff` ties us to a specific type of system. Moreover, the Kconfig-based change classification has a pervasive effect on the different components of the tool, making adaptation potentially complicated. But the approach presented in this paper could be applicable to highly variable systems having an explicit variability model, as often found in the software product line domain for instance. While the Linux kernel is not a software product line, it does have the main technical characteristics of such systems [36] hinting that our approach could be applicable in this larger context. Existing feature change classifications [8, 26] can be adapted, as we did in this work, to match other feature notations. Then, one will have to adapt the feature model comparison process to support that new classification. Previous work on feature models showed that their maintenance can be complex and error prone [5, 15]. With an approach such as `FMDiff`, it would be possible to extract new information about the evolution of the features using already existing artefacts, at the cost of adapting our tool.

Finally, the last threat to the external validity of our study concerns the Linux-specific character of the comparison of the evolution of architecture-specific FMs. While not all SPLs are affected by the hardware architecture they run on, we can often find a set of high-level features that can be used to define "sub-product lines" as we did using the architectures with the Linux kernel FM. In such cases, one can apply the methodology presented in this work to analyze the co-evolution of those different "sub-product lines". For instance, in the automotive domain, one can use this approach to identify which feature changes affected the variability model of the "sport","city" and "family" variants of a car, where each variant is a product line on its own. Such view of the effect of changes can be of use in area other than the Linux kernel.

## 7 Related work

The idea of using features as first-class entities during highly variable system development and evolution has been considered many times in the past. Using features as evolutionary units is a key concept of the feature-oriented development paradigm [4]. Existing approaches also propose to manage the evolution of large variability models by describing series of delta in terms of features [7, 45]. Finally, several studies highlight the relationship between the evolution of a SPL implementation and its feature model in open-source projects [30] and in industrial contexts [16]. While not directly related to our work, those studies exemplify the role feature changes play in the evolution of complex systems.

In the context of this work, we designed a new feature change classification scheme, similar to what can be found in other studies. In [32], Seidl proposes a classification of evolution scenarios on SPLs based on the impact of feature-changes on the mapping between features and other models (class diagram), as a mean to preserve a consistent mapping between features and model elements. Furthermore, in the work of Neves et al. on the safe evolution of SPLs [25], we can observe that the change scenarios described in this work intertwine evolution of the variability model and its implementation. Finally, in [28], Passos et al. envision that adopting a feature-oriented view on software evolution could enable easier traceability, analysis and generally facilitate evolution management. All of those studies comfort us in the idea that feature evolution is tightly coupled to the evolution of its associated product line, and as a consequence that the evolution of the feature model reflects the evolution of the product line as a whole; the main idea behind of our study.

Several FM change classifications have been proposed in the past. In his thesis, Paskevicius describes [26] several transformations that can be applied to a FM. Similarly, FM change patterns have been identified by Alves et al. in [3] and Neves et al. in [25]. In his study of the co-evolution of models and feature mapping [32], Seidl also describes a set of operations applied to FMs. Thüm et al. [44] 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. [29, 30] 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. [35] a `depends` statement can either be interpreted as a cross-tree constraint or a hierarchy relationship. As a consequence, we cannot automatically decide how a depend statement should be mapped to more standard FODA notation [18] and reuse the appropriate change classifier. Secondly, `FMDiff` is able to capture changes in feature attributes which are not considered by these classifications.

Variability models can become very large, and the complexity of the relationships between features can make the manual analysis of feature changes extremely complicated [44]. To mitigate this, several variability model differencing techniques were designed to facilitate change comprehension. In [2], Archer et al. present two differencing approaches for feature models: syntactic and semantic, suggesting that the semantic approach

would yield more actionable results than the syntactic. The syntactic approach amounts to textual differences and, in the case of Linux, this information is already available to developers through the use of the Git diffing toolset. In the semantic approach, the output can either be sets of configurations or partial feature models describing the sets of configurations that were possible before the evolution and are invalid now, or vice-versa. Although this might be possible, the number of features in the Linux kernel might be problematic for existing semantic differencing approaches [22, 24]. FMDiff performs a semi-structured diff operation: we preserve the features and their statements, and perform textual comparison at an attribute level. This approach provides additional benefits compared to textual differences since using our approach, a developer can visualize the effect of a hierarchy change on its features, and observe the spread of the changes across architectures. However, with this approach we cannot obtain the semantic differences and provide information about changes in allowed configurations and cannot express feature model changes in terms of features - we express them in terms of feature changes.

The Linux kernel has been used as an example of an evolving highly variable system many times in the past. Israeli et al. show in [17] that the Linux kernel evolution follows some of Lehman's laws of software evolution [20], namely the continuing growth by measuring the number of lines of code over time. Lotufo et al. [21] study the evolution of the Linux kernel variability model over time through FM structural metrics evolution (model size, number of leaves, etc.). They show in their study that the number of features and constraints increases 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 FM structure, properties, and evolution, several research teams have developed tools to reconstruct a FM from Kconfig files. LVAT [35] and `Undertaker` [10, 38, 42] are the main examples of such tools. We chose to rely on `Undertaker` for its convenient wrapping of `kconfigdump`, 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, `kconfigdump` flattening of the hierarchy facilitated capturing feature hierarchy changes through changes of `depends` statements.

In recent work, Passos et al. built a dataset of feature changes of Linux [27]. Focusing only on addition and removal of features, this dataset relates feature changes, commit information, and file changes. In comparison, `FMDiff` captures feature changes but does not use nor rely on commit information and file change details. We have shown that modifications played a major role in the evolution of the Linux FM, and for this reason the dataset built using `FMDiff` appears to be more suited to describe in details the evolution of the Linux FM.

## 8 Conclusion

The main contribution of our work is an approach to extract and classify changes from the history of a Kconfig-based feature model. Our approach is based on a dedicated feature change classification scheme, focused on the Kconfig language, describing feature changes at different levels of granularity. Using this classification, we can describe changes occurring on features, feature attributes, and feature attribute values.

As a second contribution, we proposed both the `FMDiff` tool, automating our approach, as well as the dataset we built during this study. We showed that the data obtained with this tool is consistent with changes observed in the Kconfig model, and provides more comprehensive information about feature changes than what could be obtained using textual differences. We used our tool to extract feature model changes occurring in sixteen releases of the Linux kernel, building a structured and detailed history of the Linux kernel FM evolution.

We used the `FMDiff` dataset to explore the evolution of the Linux kernel feature model. Our findings regarding the evolution of this model constitute our last two contributions, highlighting the informative value of fine-grained feature changes and approaches such as `FMDiff`.

We identified the most common feature change operations occurring on the Linux kernel feature model, namely modification of existing features. We suggest this might give a different orientation to future research as this type change is under-represented in current research on feature model evolution.

We also relied on `FMDiff` data to compare the evolution of the different architecture-specific FMs of the Linux kernel. This allowed us to show that the different architectures evolved differently, and that feature changes affecting multiple architectures were common. Based on this information, we made the following two observations. First, we pointed out that future research on the evolution of the Linux kernel FM should specify which architectures were studied, as observations made on a small subset of architecture-specific FMs are not generalizable to all of them without careful consideration. We then show that the gathered information al-

lows to reflect on the development practices of the kernel developers with respect to multi-architecture development processes.

We believe that the information captured by `FMDiff` can be used to facilitate maintenance operations. The dataset built using `FMDiff` could be used to link the evolution of variability models with the evolution of their implementation. Modifications of feature dependencies captured by our approach could be valuable information when observing changes in code dependencies for instance. Another possibility would be to explore the relationship between the fine-grained changes and delta-oriented approaches used in the management of product lines, where our representation of changes could be of use. While we have shown here that feature changes do not equally affect all architecture-specific feature models of the Linux kernel, a subset of the architecture-specific FMs might evolve similarly. The identification of such groups of architecture-specific FMs would allow us to refine the extent to which conclusions drawn from the observation of a single architecture-specific FMs can be generalized.

## 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.

## References

1. I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.
2. M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature Model Differences. In J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, editors, *Advanced Information Systems Engineering*, number 7328 in Lecture Notes in Computer Science, pages 629–645. Springer Berlin Heidelberg, 2012.
3. 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, pages 201–210. ACM, 2006.
4. S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.
5. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, Sept. 2010.
6. T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec. 2013.
7. G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. EvoFM: feature-driven planning of product-line evolution. In *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*, PLEASE '10, pages 24–31. ACM, 2010.
8. G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski. Towards Feature-driven Planning of Product-line Evolution. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 109–116, New York, NY, USA, 2009. ACM.
9. P. Clements and L. Northorp. *Software Product Lines, 2nd edition*. Addison-Weasley, Reading, MA, 2002.
10. C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A robust approach for variability extraction from the Linux build system. In *Proceedings of the 16th International Conference on Software Product Line*, SPLC '12, pages 21–30. ACM, 2012.
11. C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding Linux feature distribution. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, MISS'12, pages 15–20. ACM, 2012.
12. N. Dintzner, A. Van Deursen, and M. Pinzger. Extracting feature model changes from the Linux kernel using FMDiff. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14. ACM Press, 2013.
13. 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 at Runtime*, volume 509 of *MRT 2009*, pages 61–70, 2009.
14. E. Giger, M. Pinzger, and H. Gall. Can we predict types of code changes? An empirical analysis. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR'12, pages 217–226. ACM, June 2012.
15. J. Guo, Y. Wang, P. Trinidad, and D. Benavides. Consistency maintenance for evolving feature models. *Expert Systems with Applications*, 39(5):4987–4998, 2012.
16. R. Hellebrand, A. Silva, M. Becker, B. Zhang, K. Sierszecki, and J. Savolainen. Coevolution of Variability Models and Code: An Industrial Case Study. In *Proceedings of the 18th International Software Product Line Conference*, volume 1 of *SPLC '14*, pages 274–283, New York, NY, USA, 2014. ACM.
17. A. Israeli and D. G. Feitelson. The Linux kernel as a case study in software evolution. *J. Syst. Softw.*, 83(3):485–501, 2010.
18. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.
19. A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.
20. M. Lehman. Laws of software evolution revisited. *Software process technology*, pages 108–124, 1996.
21. R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. 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.
22. S. Maoz, J. O. Ringert, and B. Rumpe. A Manifesto for Semantic Model Differencing. In J. Dingel and A. Solberg, editors, *Models in Software Engineering*, number 6627 in Lecture Notes in Computer Science, pages 194–203. Springer Berlin Heidelberg, 2011.

23. S. Nadi and R. Holt. Mining Kbuild to Detect Variability Anomalies in Linux. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 107–116, 2012.
24. T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
25. 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.
26. 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.
27. L. Passos and K. Czarnecki. A Dataset of Feature Additions and Feature Removals from the Linux Kernel. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 376–379, New York, NY, USA, 2014. ACM.
28. L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. Feature-oriented Software Evolution. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 17:1–17:8, New York, NY, USA, 2013. ACM.
29. 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 Feature Oriented Software Development*, FOSD '12, pages 62–69. ACM, 2012.
30. L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wasowski, 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 2013, pages 91–100. ACM, 2013.
31. D. Romano and M. Pinzger. Analyzing the Evolution of Web Services Using Fine-Grained Changes. In *Proceedings of the 19th International Conference on Web Services*, ICWS '12, pages 392–399, 2012.
32. 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 of *SPLC '12*, pages 76–85. ACM, 2012.
33. S. She and T. Berger. Formal Semantics of the Kconfig Language. Technical Note, University of Waterloo, Waterloo (ON) Canada, 2010.
34. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The Variability Model of The Linux Kernel. *VaMoS*, 10:45–51, 2010.
35. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461 –470, 2011.
36. 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 '07, page 30, 2007.
37. J. Sincero and W. Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. *SPLC*, pages 257–260, 2008.
38. 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.
39. H. P. Siy and D. E. Perry. Challenges in Evolving a Large Scale Software Product. In *Proceedings of Principles of Software Evolution Workshop at the International Software Engineering Conference*, ICSE '98, pages 251–260, 1998.
40. M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. Muller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of the Third Working Conference on Reverse Engineering*, WCRE '96, pages 31–40, Nov. 1996.
41. M. Svahnberg. *Variability in Evolving Software Product Lines*. PhD thesis, Research Board at Blekinge Institute of Technology, 2000.
42. 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 6th Conference on Computer systems*, EuroSys '11, pages 47–60. ACM, 2011.
43. 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*, FOSD '09, pages 81–86, 2009.
44. T. Thuem, D. Batory, and C. Kaestner. Reasoning about edits to feature models. In *Proc. of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264. IEEE Computer Society, 2009.
45. J. White, J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt. Evolving Feature Model Configurations in Software Product Lines. *J. Syst. Softw.*, 87:119–136, Jan. 2014.
46. A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Trans. Softw. Eng.*, 30(9):574–586, Sept. 2004.

TUD-SERG-2015-004
ISSN 1872-5392