Fever: Extracting feature-oriented changes from commits

Dintzner, Nicolas; Van Deursen, Arie; Pinzger, Martin

DOI
10.1145/2901739.2901755

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016

Delft University of Technology
Software Engineering Research Group
Technical Report Series

FEVER: Extracting Feature-oriented Changes from Commits

Nicolas Dintzner, Arie van Deursen, Martin Pinzger

Report TUD-SERG-2016-005

TU Delft

SERG

TUD-SERG-2016-005

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 *Proceedings of the 13th International Conference on Mining Software Repository*

FEVER: Extracting Feature-oriented Changes from Commits

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

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

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

ABSTRACT

The study of the evolution of highly configurable systems requires a thorough understanding of thee core ingredients of such systems: (1) the underlying variability model; (2) the assets that together implement the configurable features; and (3) the mapping from variable features to actual assets. Unfortunately, to date no systematic way to obtain such information at a sufficiently fine grained level exists.

To remedy this problem we propose FEVER and its instantiation for the Linux kernel. FEVER extracts detailed information on changes in variability models (KConfig files), assets (preprocessor based C code), and mappings (Makefiles). We describe how FEVER works, and apply it to several releases of the Linux kernel. Our evaluation on 300 randomly selected commits, from two different releases, shows our results are accurate in 82.6% of the commits. Furthermore, we illustrate how the populated FEVER graph database thus obtained can be used in typical Linux engineering tasks.

CCS Concepts

•**Software and its engineering** → **Model-driven software engineering;** *Feature interaction; Software design engineering;*

Keywords

highly variable systems, co-evolution, feature, variability

1. INTRODUCTION

Highly configurable software systems allow end-users to tailor a system to suit their needs and expected operational context. This is achieved through the development of *configurable* components, allowing systematic reuse and mass-customization. [1]. Examples of such systems can be found in various domains such as database management [2,3], SOA based systems [4], operating systems [5], and a number of industrial [1] and open source software projects [6] among which the Linux kernel may be the most well-known.

In the implementation of such system, configuration options, or *features*, play a significant role in a number of inter-related artefacts of different nature. For systems where variability is mostly resolved at build-time, features will play a role in, at least, the following three spaces [7, 8]:

1. *the variability space* - describing available features and their allowed combinations;
2. the *implementation space*, comprised of re-usable assets, among which configurable implementation artefacts; and finally
3. the *mapping space* - relating features to assets and often supported by a build system like Makefiles;

When such systems evolve, information about feature implementation across those three spaces is actively sought by engineers [9]. Inconsistent modifications across the three spaces (variability, mapping, and implementation) may lead to the incapacity to derive products, code compilation errors, or dead code [10–12]. Consistent co-evolution of artefacts is a necessity adding complexity to an already non-trivial evolutionary process [13], occurring in both industrial [14] and open-source contexts [15, 16].

Recent studies [7, 15] described common changes occurring in such systems, giving insight on how each space could evolve, and revealing the relationship between the various artefacts. More recently, Passos et al. proposed a dataset capturing the addition and removal of features [17].

Such feature-related change information is important in various practical scenarios.

- A release manager is interested in finding out which commits participated in the creation of a feature, to build the release notes for instance. In such cases, he would be interested in commits introducing the feature, and the following ones, adjusting the behaviour or declaration of the feature.
- A developer introducing a new feature to a subsystem will be interested in finding how such feature was supported by similar subsystems in the past. Then, (s)he needs to look for changes in those subsystems, involving that feature.

[1] http://splc.net/fame.html

MSR '16 May 17–18, 2016, Austin, TX, USA

DOI: 10.1145/1235

- Researchers focusing on feature-oriented evolution of systems are interested in automatically identifying instances of co-evolution patterns or templates, or extending the existing pattern catalogue.

Unfortunately, the most detailed change descriptions currently available [7, 15] were obtained using extensive *manual* analysis of commits, and the existing datasets do not provide the necessary links between features and associated assets to enable such queries.

To remedy this problem, we present FEVER (Feature EVolution ExtractoR), a tool-supported approach designed to automatically extract changes in commits affecting artefacts in all three spaces. FEVER retrieves the commits from a versioning system and rebuilds a model of each artefact before and after their modification. Then it extracts detailed information on the changes using graph differencing techniques. Finally, relying on naming conventions and heuristics the changes are aggregated based on the affected feature(s) across all commits in a release. The resulting data is then stored in a database relating the features and their evolution in each commit.

While the tool we built to extract changes is centred on the Linux kernel, the approach itself is applicable to a wide set of systems [16,18] with an explicit variability model, where the implementation of variability is performed using annotative methods (pre-processor statements in our case), and where the mapping between features and implementation assets can be recovered from the build system.

With this study, we make the following key contributions: (1) a model of feature-oriented co-evolving artefacts, (2) an approach to automatically extract instances of the model from commits, (3) a dataset of such change descriptions covering 5 releases of the Linux kernel history (3.11 to 3.15 in separate databases), (4) an evaluation of the accuracy of our heuristics showing that we can extract accurately the information out of 82.6% of the commits, (5) we show how the FEVER dataset can be used to assist developers and researchers in performing the aforementioned tasks, and finally, the tool and datasets used for this study are available on our website.[2]

We first provide information on previous work on the evolution of highly variable systems in Section 2. We then give additional information on how variability can be implemented using the Linux kernel as an example in Section 3. Then, we present the feature-oriented change model we use to describe the evolution of such systems in Section 4. We explain the main steps of the model-based change extraction process in Section 5. We evaluate our prototype implementation of FEVER by manually validating a subset of 300 randomly selected commits we extracted from release v3.11 and v3.12 of the Linux kernel and present the results in Section 6. Finally in Section 6.3, we discuss the possibilities and limitations of our approach, and elaborate on its usage in the context of complex change description and configurable software maintenance operations in Section 7.

2. RELATED WORK

Variability implementation in highly-configurable systems has been extensively studied in the past [19]. While many approaches can be found to analyze features in each individual space, few focus on their detailed evolution or the consolidation of such changes.

In [20], we introduced FMDiff, an approach to extract feature model changes, that we reuse for the approach presented in this paper. In this work, we extend FMDiff concepts to cover all types of artefacts and relate those changes on a feature-basis.

Several studies present methods to extract variability information from build systems [21–23]. Such approaches are designed to study the current state of the system, and require all files to be present. In our case, we are interested by the changes as performed by developers, focusing on commits which avoid the need for a costly (and often impossible) analysis of the entire build system. We built a custom Makefile parser allowing us to extract information relying on modified artefacts only.

Variability implementation using annotative methods in source file were also studied in the past [24], often for error detection [10, 25, 26]. In this study, we use the approach presented in [6] to identify code blocks and their condition, and we then rely on this representation to build a model of implementation assets.

Only few studies focused on the co-evolution of artefacts in all three variability spaces: variability model (VM), mapping, and implementation. In [7], Neves et al. describe the core elements involved in feature changes (VM, mapping, and assets). A collection of 23 co-evolution patterns is presented by Passos et al. in [15]. Each pattern describes a combination of changes that occur in the three variability spaces. These papers aimed at identifying common change operations, and relied on manual analysis of commits. In this work, we relied on such change descriptions to design the FEVER change meta-model, and we focused on how to extract automatically such changes.

Change consolidation across heterogeneous artefacts has been a long standing challenge. For instance, Begel et al. proposed a large database aggregating code level information, people, and work items [27]. We take a different approach, and propose to extract more detailed information focusing on implementation artefacts only. Recently, Passos et al. created a database of feature addition and removal [17] in the Linux kernel. We extend this work by extracting detailed changes on *all* commits, and provide such descriptions on *all* types of artefacts. The FEVER dataset is, to the best of our knowledge, the first dataset providing a consolidated view of complex feature changes.

3. BACKGROUND

In this section, we present how the variability is supported in the Linux kernel, the different artefacts involved in its realization and their relationships.

3.1 Variability Model

A variability model (VM) formalizes the available configuration options (which we assimilate to "features" in this work) of a system as well as their allowed configurations [28]. In the context of the Linux kernel, the VM is expressed in the Kconfig language. An example of a feature as described the Kconfig language shown in Listing 1. 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, which may be "boolean" (selected or not), "tristate" (selected, selected but compiled as a module,

[2] http://swerl.tudelft.nl/bin/view/NicolasDintzner/

or not selected), or a value (when the type is "int", "hex", or "string"). In our example the SQUASHFS_FILE_DIRECT feature is of type *boolean* (line 2). In the remainder of this work, we will refer to boolean and tristate features simply as "boolean features", while features with type "int", "hex", or "string", will be referred to as "value-based features". The text following the type on line 3 is the "prompt" attribute. Its presence indicates that the feature is visible to the end user during the configuration process. Features can also have default values. In our example the feature is selected by default (*y* on line 5). The default value might be conditioned by an "if" statement.

Kconfig expresses feature dependencies using the "depends" statements (see line 5). If the expression is satisfied, the feature becomes selectable during the configuration process. In this example, the feature SQUASHFS must be selected. Reverse dependencies are declared using the "select" statement. If the feature is selected then the target of the "select" will be selected automatically as well (ZLIB_INFLATE is the target of the "select" statement on line 6). The selection occurs if the expression in the following "if" statement is satisfied by the current feature selection (e.g., if SQUASHFS_ZLIB is already selected).

In the context of this study, we consider additions and removals of features as well as modifications of existing ones i.e., modifications of any attributes of a feature.

```
1 config SQUASHFS_FILE_DIRECT
2    bool
3    prompt "Decompress files in page cache"
4    default y
5    depends on SQUASHFS
6    selects ZLIB_INFLATE if SQUASHFS_ZLIB
7    help
8      Decompress file data in page cache.
```

Listing 1: A feature declaration in Kconfig

To create a new kernel image, an end-user uses a configurator tool ("menuconfig" for instance) which reads the variability model, and present the features to the user in a tree like structure. At the end of the configuration process, a list of selected features is passed on to the build system which uses it to select artefacts and artefact fragments to include in the image before compiling them.

3.2 Feature-asset Mapping

The mapping between features and assets determines which assets should be included in a product upon the selection of specific features. In highly-configurable systems, the assets could be source code, documentation, or any other type of resources (e.g., images). In the context of this study, we focus on implementation artefacts. The addition of the mapping between a feature and code in a Makefile, as performed in the Linux kernel, is presented in Listing 2.

Upon feature selection, the name of the feature used in the Makefile (symbol prefixed with CONFIG_) will be replaced by its value. As a result, the compilation units (".o" files) will be added to different lists "obj-y", "obj-n", and "obj-m" (for modules), based on the value of the macros CONFIG_SQUASHFS_FILE_DIRECT. Compilation units added to the list "obj-y" are compiled into the kernel image while those in "obj-m" are compiled as external modules, and objects in "obj-n" are not compiled.

Alternatively, a developer may chose to include directly "obj-y" list in his Makefile, in which case, the content of the list will be included in the compilation process as soon as the Makefile is included in the build process. The inclusion of a Makefile in the build process may be subject to feature selection.

```
1 + obj-$(CONFIG_SQUASHFS_FILE_DIRECT) +=
2 +        file_direct.o page_actor.o
```

Listing 2: Mapping between features and assets as performed in the Linux kernel

The language used to describe the mapping and implement the compilation process is a complete programming language, and the exact mapping between feature and assets can be very complex. Makefiles are organized in a hierarchy, and constraints from one may affect others, leading to complex presence condition for artefacts.

3.3 Assets

Many types of assets exists, such as images, code, or documentation. We consider only configurable implementation assets (source files). We focus specifically on preprocessor based variability implementation (using #ifdef statements), which, despite known limitations [29], is still widely used today [6]. An example of an addition of a preprocessor statement is presented in Listing 3 where feature SQUASHFS_FILE_DIRECT is used to condition the compilation of two code blocks, one pre-existing (line 2 to 7) and a new one (lines 9 to 13). As a result, based on the selection of the feature SQUASHFS_FILE_DIRECT during the configuration phase, only one of the two code blocks will be included in the final product.

```
1 + #ifndef CONFIG_SQUASHFS_FILE_DIRECT
2  static inline void *squashfs_first_page
3        (struct squashfs_page_actor *actor)
4  {
5   return actor->page[0];
6  }
7 + #else
8 + static inline void *squashfs_next_page
9 +      (struct squashfs_page_actor *actor)
10 + {
11 + return actor->squashfs_next_page(actor);
12 + }
13 +#endif
```

Listing 3: Creating an #ifdef block in Linux

Value-based features will be referenced in the implementation, acting as a place-holder for a value defined during the configuration process, as shown in Listing 4.

```
1 //set the value of "DSL"
2 #define DSL  CONFIG_DE2104X_DSL
```

Listing 4: Referencing a value feature

4. DESCRIBING CO-EVOLUTION: THE FEVER CHANGE META-MODEL

The objective of this work is to obtain a consolidated view of changes occurring to features and their implementation. We present in this section the meta-model we use to describe feature-related changes to individual artefacts, and how we relate those changes to one-another. We illustrate the us-

age of the model with a example of actual feature changes, affecting all spaces, extracted from release v3.11. In this scenario, a developer commits a new driver for an ambient light sensor, "APDS9300".

4.1 FEVER co-evolution change meta-model

An overview of the FEVER change meta-model is shown in Figure 1. This overview highlights the different entities we use to describe what occurs in a commit, from a feature perspective.

The **commit** represents a commit in a version control system. **Commit** entities are related to one another through the "next" relationship, capturing the sequence of changes over time. Each **commit** "touches" a number of artefacts, and those changes are captured in **ArtefactEdit** entities. The **commit** may affect any of the three spaces, leading to **SourceEdit** entities when features are modified at a source level, **MappingEdit** entities when the mapping between feature and assets is affected, or finally **FeatureEdit** entities when the variability model changes. While the **ArtefactEdit** indicates a change to a file, **Source-, Mapping- and Feature- Edit** entities are all representing the change related to individual features within those files. We omitted the following relationship in the model for readability purposes: **FeatureEdit**, **MappingEdit**, and **SourceEdit** entities are linked to **ArtefactEdit** with a "in" relationship, pointing to the artefact in which the change took place.

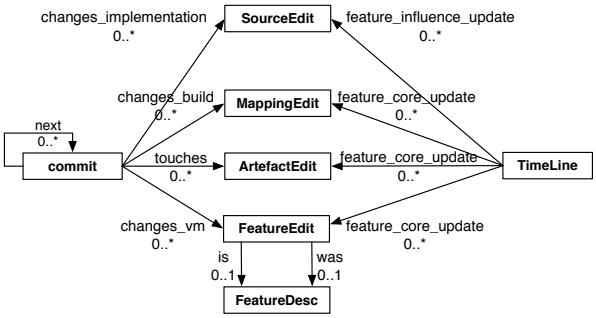

Figure 1: FEVER Feature-oriented change model

For a commit in the repository we record the commit id (sha1) to relate our data with the original repository. We save the commit message which may contain information about the rationale of a change. Finally, to keep track of who touches which feature, we record people-related information such as commiter and author of each commit.

4.2 Variability model changes

A **FeatureEdit** entity represents the change of one feature within the variability model performed in the context of a **commit**. We are interested in the affected feature, as well as the change operation that took place (*addition*, *removal* or *modification* of an existing feature). The **FeatureEdit** entity also points to a more complete description of the feature, **FeatureDesc** entities. **FeatureDesc** presents the feature as it "was" before the change (if existing) and how it "is" after the edit operation (if existing). Those entities contain the details of the feature before and after the change. From an evolution perspective however, we are more interested in the change affecting the feature, as this may be linked to changes in other spaces.

In our example presented in Figure 2 we can see on the left hand side the commit sequence, where commit "03eff" "touches" four **ArtefactEdits** (in gray), and "changes the vm" by adding a feature (in light pink). The **FeatureEdit** entity points, via the "in" relationship, to the Kconfig file in which the feature was touched. We can also see how the **FeatureEdit** entity is connected to a **FeatureDesc** (in purple) using the "is" relationship. The feature is added, as noted on the **FeatureEdit** entity.

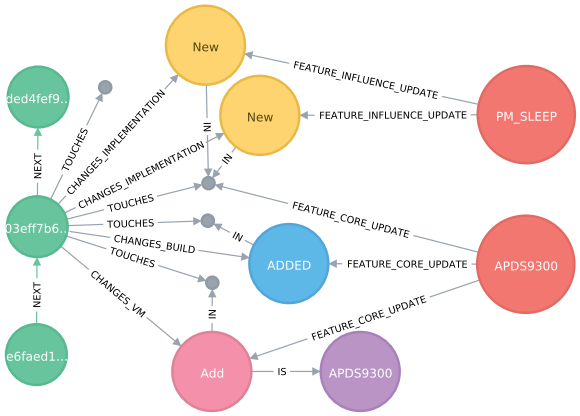

Figure 2: Change model instance for the introduction of a new driver in the Linux kernel

4.3 Mapping changes

Regarding the evolution of the mapping, we are mainly interested in the evolution of the mapping between feature and asset, in order to assign code changes, occurring within files, to features. The evolution of the mapping space is represented by **MappingEdit** entities characterized by: the feature involved, and the type of artefacts it is mapped to. We describe the feature-mapping change operation (*added*, *removed*, or *modified*), referring to the association of a feature any assets, and the change affecting the target within that mapping (*added* or *removed*). We can thus make the difference between a situation where a new mapping is introduced (*addition* of a mapping with an *added* target) and an existing mapping being extended (*modification* of a mapping with an *added* target). In the example, the **MappingEdit** entity is highlighted in blue. It is connected to the commit with a "changes_build" relationship.

4.4 Source changes

Feature related changes within source code, such as modifications to conditionally compiled blocks and feature references, are captured as **SourceEdit** entities. Feature in #ifdef code block conditions and feature references within a given file are an indication that the behaviour of the feature mapped is configurable, and its exact behaviour is determined by other features.

Feature references are references to feature names within the code, meant to be replaced by the feature's value at compile-time. Such references may only be *added* or *removed*. In such cases, the **SourceEdits** entity contains the name of the affected feature and the change in question.

Conditionally compiled blocks are identified by the conditions under which they will be included in the final product. A change to such block is represented by a **SourceEdit** containing the exact condition of the block, the change to the block itself (*added, removed, modified*), and the change of the implementation within that block: *added* if the code is entirely new, *removed* if the whole block was removed, *modified* when the changed block contains arbitrary edits, or finally *preserved* if the code itself has not been touched.

In our example, two **SourceEdit** entities, in yellow in Figure 2, are connected to the commit indicating that the commit affected conditionally compiled blocks, and to the file "in" which those changes occurred.

4.5 TimeLines: Aggregating feature changes

Changes pertaining to the same features are then aggregated into **TimeLine** entities. For this study, we created **TimeLine** entities for entire releases.

We divide the types of changes that may affect a feature into two broad categories: *core changes* and *influence change*. A *feature core change* indicates that the behaviour of the feature itself or its definition is being adjusted. This comprises changes to the feature definition in the VM, changes to the mapping between the feature and assets, and changes affecting assets mapped to that feature. A *feature influence change* indicates that the feature is playing a role in the behaviour of another feature. This is visible in a **SourceEdit**, through reference of that feature in conditionally compiled code blocks, as part of a condition, or referenced for its value.

In Figure 2, two **TimeLine** entities are depicted in red. The first one relates to the feature that was introduced. We can see that the "APDS9300" node is connected to the **FeatureEdit**, the **MappingEdit** and an **ArtefactEdit** with a "feature_core_update" relationship. The connection between the **TimeLine** for this feature and the **ArtefactEdit** is deduced from the **MappingEdit**: because the new mapping assigns this artefact to feature APDS9300, then the introduction of this artefact is a "core" update of this feature. The APDS9300 **TimeLine** connects the different changes occurring in 3 different types of artefacts, all related to the same operation: the addition of a feature.

Moreover, we can see that a **TimeLine** for feature PM_SLEEP is present and connected to two **SourceEdit** entities. This indicates that, at the creation time, the driver APDS9300 interacts with the power management "sleep" feature, and this interaction occurs in two different code blocks.

It is important to note that changes are extracted on an "per artefact basis". This means that entities being moved within the same artefacts (a feature in a Kconfig file, or a mapping in Makefile) will be seen as modified. However, if an entity is moved from one artefact to another, this is captured as two separate operations: a removal and an addition, and as such, two **Edits** entities. Those two **Edit** entities are linked together by a **TimeLine** entity, referring to the modified feature.

5. POPULATING FEVER

5.1 Overview

The FEVER approach starts from a set of commits and outputs an instance of the FEVER change model covering the given commit range. Figure 3 presents an overview of the change extraction process.

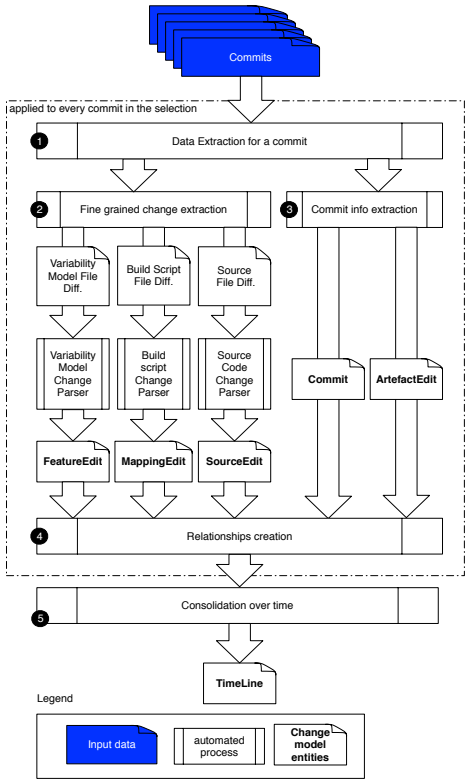

Figure 3: Overview of the FEVER change extraction and consolidation process

From the initial set of commits, FEVER first analyses each commit separately, and then consolidates the extracted change information. For each commit, steps 1 to 4 are executed as follows:

Step 1 is the identification of the touched artefacts and the dispatch to the appropriate change parser. In the Linux kernel, artefact types are characterized by naming conventions and file extensions: "Kconfig" for VM files, "Makefile" or "Kbuild" and "Platform" for build files, and ".c",".h", ".s", ".dts", and ".dtb" for source code. Note that ".dts" and ".dtb" files also contain C code with pre-processor statements.

Step 2 performs the artefact-specific data extraction processes. The next subsections detail the process for each type of artefact, but all of them follow the same general steps. First FEVER rebuilds a model of the artefact as it was before the change, and a second one representing the same artefact after the change. Then, FEVER uses the EMF Compare[3] infrastructure to identify the differences between the two versions of the model. EMF Compare identifies the differences between the two models, and extracts them in terms of the EMF meta-model. FEVER then translates those changes into the different **Edit** entities depending on the artefact type.

The reconstruction of the models, and the identification of changes (based on EMF Compare results) are based on heuristics and assumptions on the structure of the artefacts.

[3] `http://wiki.eclipse.org/EMF_Compare`

We provide an estimation of the accuracy of those heuristics in Section 6.

Step 3 is the extraction of changes in artefacts for which we do not extract detailed changes. This includes only commit-related information from which we create a **commit** entity, and "untyped" artefacts (documentation, scripts...), represented by **ArtefactEdit** entities.

In *Step 4*, we create the relationships between **Edit** entities, the **Commit**, and **ArtefactEdit**.

Step 5 of our approach consists in creating entities and relationships spreading beyond single commits: "next" relationships among commits, and feature **TimeLine** entities with their respective relationships to edit entities. This is done by running through every commit, and identifying touched feature(s), creating if necessary a new **TimeLine** entity and the appropriate relationships between the **TimeLine** and relevant edits.

5.2 Extracting Variability Model Changes

The characteristics of the changed features that we focus on are their type (boolean or value-based), their visibility, and their optionality as described in Section 3.

We first reconstruct two instances of the VM depicted in Figure 4-A per VM file touched, one representing the VM before the change, the other after the change. If, like in the case of the Linux kernel, the VM is described in multiple files, we reconstruct the parts of the model described in the touched files, i.e., the model we rebuild is always partial. The extraction process follows the FMDiff approach [20], including the usage of "dumpconf". This tool takes as an input a Kconfig file and translates it into XML. "dumpconf" is designed to work on the complete Kconfig model, where the different files are linked together with a "source" statement, similar to #include in C. To invoke "dumpconf" successfully on isolated files, we remove the "source" statements as a pre-processing steps. "dumpconf" also affects the attributes of features, and the details of the change operation are described in [30]. We use this XML representation of the Linux VM to build the model shown in Figure 4-A.

We then use EMF Compare to extract the differences and compile the information in a **FeatureEdit** entity. We attach to this entity the snapshot of the feature as it was before and after the change in **FeatureDesc** entities. If the feature is new, respectively deleted, we do not create a "before", respectively "after", **FeatureDesc** entity.

5.3 Extracting Build Changes

Similarly to the extraction of VM changes, **MappingEdit** entities are created based on the differences of reverse engineered models of a Makefile, before and after the change. We use the model shown in Figure 4-B.

The model contains a set of features and symbols mapped to targets. "Symbol" refers to any variable mapped to any assets which is not a feature. We identify feature names in Makefiles by their prefix "CONFIG_". We scan the Makefiles and extract pairs of symbols by searching for assignment operators ("+=" and ":="). We consider that the symbol on the left hand side is mapped to the symbol on the right hand side (target).

To determine the type of a targeted asset, we use the following rules: Compilation unit names finish with either ".o",".dts", ".dtb"; compilation flags contain specific strings ("cc-flags", "-D", "-L", "-m", or "-W"). We identify folder

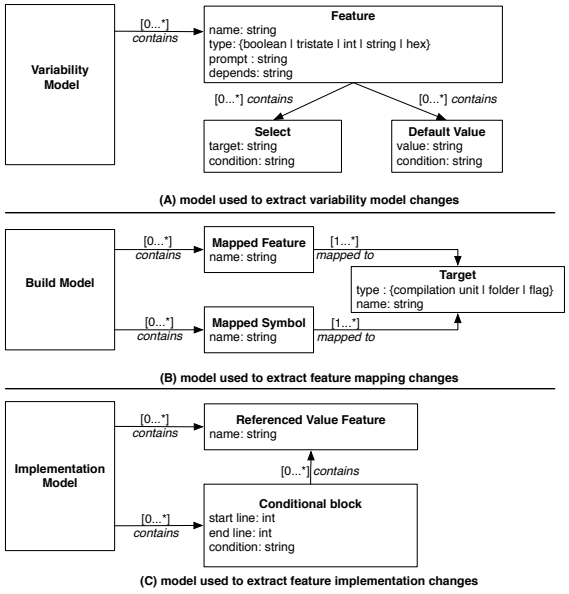

Figure 4: Variability, Mapping and Implementation models used for change extraction

names by "/", or single words, not containing any special characters nor spaces. When features are found as part of "ifeq" or "ifneq" statements, we consider that they are mapped to any targets contained within their scope. In Listing 5, both CONFIG_OF and CONFIG_SHDMA will be mapped to the compilation unit "shdma.o".

We also resolve aliases within Makefiles. An example of an alias is presented in Listing 5, where feature TREE_TEST is mapped to the alias "tree_test.o" referring to two compilation units "tree_main.o" and "tree.o". This step is performed as a post-processing step for each build model instance, and is based on heuristics, also evaluated in Section 6.

```
1 ifeq ($(CONFIG_OF),y)
2     shdma-$(CONFIG_SHDMA) += shdma.o
3 endif
4 obj-$(CONFIG_TREE_TEST) += tree_test.o
5 tree_test-objs := tree_main.o tree.o
```

Listing 5: Example of an "ifeq" statement and aliases used in Makefiles

We then use EMF Compare to extract the differences between the two model instances, giving us the list of feature mappings that were added or removed in that commit.

As mentioned in Section 3, the exact mapping between features and files is the result of a complex Makefile hierarchy. By focusing on the mapping as described in a single Makefile, FEVER only captures a part of the presence condition of each file.

5.4 Extracting Implementation Changes

At the implementation level, we consider changes to #ifdef blocks and changes to feature references in the code, as presented in Section 3. To extract those changes, we rebuild a model of each implementation file in its before and after state following the model presented in Figure 4-C.

To rebuild the models, we rely on CPPSTATS [6] to obtain starting and ending lines of each #ifdef block as well as their guarding condition. It should be noted that CPPSTATS expends conditions of nested blocks within a file, facilitating the identification of block conditions. In the model, code blocks and their #else counter-parts are captured as two distinct entities. "Referenced value features" are obtained by scanning each modified source file looking for the usage of the "CONFIG_" string outside of comments and #ifdef statements.

We then use EMF Compare to compare the two models and build the **SourceEdit** entities. We determine the code changes occurring inside #ifdef blocks to compute the value of the "code edit" attribute of **SourceEdit** entities. We extract from the commit the diff of the file in the "unified diff" format, and identify which lines of code where modified. We compare this information with the first and last lines of each modified code block to determine which code block is affected by the code changes.

5.5 Change consolidation and TimeLines

The final step consist in the creation of feature **TimeLine** entities, and relate them to the appropriate entities. We create such entities for every feature touched affected by any change in any **Edit** entity. We apply the following rules:

- if a feature is touched in the VM, mapping or source file, the corresponding **Edit** entity is associated with a **TimeLine**;
- if a **SourceEdit** changes a block condition, the source edit is connected to one **TimeLine** entity per feature present in the condition;
- if an artefact is touched, it is linked to the **TimeLine** entity of the feature(s) to which it is mapped;

In order to map file changes to features, we need to know the mapping between features and files. Note that FEVER only focuses on mapping changes, leaving us with a gap with respect to mappings that are not touched. As a result, many files, whose mapping has not evolved would not be mapped - wrongly - to any features. To compensate for this, we create a snapshot of the complete mapping based on the state of the artefacts on the first commit of the commit set. This is the only operation we perform requiring the entire code base. We then run through all commits, starting from the leaves in a breadth-first manner, creating or updating **TimeLine** as necessary, and updating the known mapping between files and features as we encounters **MappingEdits**.

Some files in the Linux kernel cannot be mapped to directly to features. This concerns mostly header files, contained in "include" folders. "Include" folders do not contain Makefiles, which prevents direct mapping between features and such artefacts. Moreover, such files are included in the compilation process on the basis that they are referenced by implementation files (#include statement), which by definition bypasses any possible feature-related condition. For those reasons, we do not attempt to map such files to features. They are, however, highly conditional, and often contain many #ifdef statements, which we track.

6. EVALUATING FEVER WITH LINUX

The FEVER change extraction process is based on heuristics, and assumptions about the structure of the artefacts. Those heuristics affect the model build phase, and the comparison process - the mapping between EMF model changes and higher-level feature oriented changes. It is then important to evaluate whether the data captured by FEVER reflects the changes that are performed by developers in the source control system, leading us to formulate the research question driving this evaluation:

> RQ: To what extent does FEVER data match changes performed by developers ?

To answer this question, we apply FEVER to two releases of the Linux kernel, and compare the changes captured by FEVER and the commits obtained from the Linux SCM (Git).

6.1 Evaluation Method

The objective is to evaluate the accuracy of the heuristics and the model comparison process used for artefact change extraction and the change consolidation process. To do so, we manually compare the content of the FEVER dataset with the information that can be obtained from Git, using the GitK user interface. The evaluation was performed by the main author of this paper.

For a set of commits, we check that the different **Edit** entities and their attributes can be explained by the changes observed in Git. Conversely, we ensure that feature-related changes seen in Git have a FEVER representation. At variability model level, we check whether the features captured by FEVER as added, removed or modified are indeed changed in a similar fashion in the Linux Kconfig files.

Regarding mapping changes, we check that the pairing of features and files is accurate and that the type of targeted artefact is also correct. Special consideration is given to the validation of the mapping between features and assets. The mapping between features and files may be the results of complex Makefile constructs and may be distributed over several files through inclusion mechanism. FEVER only considers changes on a file level, and so is unlikely to resolve such complex constructs. Whenever we are able to manually assign a file to a feature by looking only at the content of makefiles - including the Makefile hierarchy, we assume that FEVER should have the information as well. This includes cases were files are assigned to "obj-y" lists, and the mapping is done in a parent Makefile. FEVER does not capture those structures, but the mapping exists.

At the code level, we check that the blocks seen as touched are indeed touched, and we compare the condition of each block. Then, by inspecting the patch, we can see if the code changes within the blocks are correct.

Regarding **TimeLine** entities, we do no check whether all relevant changes in all commits are indeed gathered into **TimeLine** object. We make the assumption that if **TimeLine** entities are properly linked in the commits we check, then the algorithm is correct, and the check on the complete release is unnecessary. We also keep track of the commits for which all extracted information is accurate, giving us an overview of the accuracy on a commit basis.

Using FEVER, we extracted feature changes from release 3.12 and 3.13 of the Linux kernel, and randomly extracted 150 commits from each release. The selection of commits in each release was performed as follows: we randomly selected

Attribute	Population	Precision (%)	Recall (%)
VM operations			
change: *added*	208	100	100
change: *removed*	73	100	100
change: *modified*	140	80	100
Mapping operations			
target: *folder*	17	100	94
target: *compilation unit*	437	100	98
target: *compilation flag*	10	67	60
mapping change: *added*	278	99	97
mapping change: *removed*	84	100	95
mapping change: *modified*	98	100	98
target change: *added*	326	99	97
target change: *removed*	133	100	97
file-feature mapping	622	81	97
Source operations			
block change: *added*	381	81	97
block change: *removed*	229	100	99
block change: *modified*	237	97	99
code change: *added*	365	99	97
code change: *removed*	195	99	99
code change: *edited*	237	96	99
code change: *preserved*	46	32	83
reference change: *added*	6	100	83
reference change: *removed*	7	88	100
TimeLine	743	93	98

Table 1: FEVER change extraction accuracy

50 commits touching at least the variability model, 50 among the commits touching at least the mapping, and 50 touching at least source files. Those three sets are non-overlapping. So the creation of three different sets ensures that our random sample covers at least all three spaces. During the evaluation, we ignored commits associated with merges and tagged releases.

6.2 Results

The results are compiled in Table 1. The table is divided into three sections, each presenting the precision and recall of FEVER when capturing detailed changes in each of the three spaces. We then present in the last section of the table the accuracy of the **TimeLine** aggregation process.

In addition to the information contained in the table, we kept track of the commits in which changes were accurately described by the FEVER change model. Among the 300 commits studied for this evaluation, we found that FEVER extracted all change attributes accurately in 82.6% (248 out of 300) of the cases.

As shown by the numbers, our implementation of FEVER extracted the changes occurring in the variability model space precision and recall of at least 80%. In some cases, features are defined multiple times within the same file and those will be seen as modified even if they are not - hence the precision of only 80% for feature modification. This is a side effect of using model comparison, where each entity of the compared models must be uniquely identified.

Regarding the mapping space, the approach is quite successful in identifying changes to features mapped to files and folders, determining whether the mapping is new for that feature, and if the target is added or removed. However, we note that detection of features linked to compilation flags is harder. Such situations are less frequent than mapping to other types of assets, making small errors having a large impact on the statistical results. The parsing of complex Makefiles tends to lead to miss-interpretation of variables, considering them wrongly as compilation flags.

Regarding implementation changes, our heuristic is good at determining whether conditionally compiled code blocks are added or removed, with a precision of 80% or more and a recall of at least 97%. The combination of CPPSTATS and model differencing proved to be efficient to identify conditionally compiled code block changes. Certain types of code changes within the blocks are well identified: blocks with fully added, removed or modified code are captured with an accuracy of 90% or more. Similarly to what occurs at a VM level, FEVER returned a number of false positive changes with "preserved code". This occurs when a file contains multiple code blocks with the exact same condition and the exact same code. In our random sample, multiple commits edited the same files containing such structures. Considering the changes with that characteristic are not frequent, those false positives reduced drastically the measured precision, but we still have a high recall.

> The results showed that the data collected by FEVER matches the changes performed by developers in 82.6% or more of the commits.

6.3 Threats to Validity

Internal validity. To extract and analyze feature-related changes, FEVER uses model-based differencing techniques. We first rebuild a model of each artefact, and then perform a comparison. The construction of the model relies on heuristics, which themselves work based on assumptions on the structure of the touched artefacts - whether they be code, models, or mapping. For this reason, information might be lost in the process. To guarantee that the data extracted by FEVER do match what can be observed in commits, we performed a manual evaluation, covering every change attribute we consider. The evaluation showed that a large majority of the changes are captured accurately, with a precision and recall of at least 80%. This gives us confidence in the reliability of the data.

The identification of compilation flags mapped to features and changes to conditional blocks preserving the code is not captured as accurately as the other attributes. Those are the results of false positives occurring when the compared models contained duplicated entities (two code blocks with the same condition and same code for instance). Those situations are not frequent, as shown in our random sample. But because in our random sample actual changes to compilation flags and changes to blocks preserving the implementation are rare, such false positives skew the statistical results. Given the high precision and recall we obtain on all other attributes, we believe this does not affect the validity of the data.

Mapping between feature and files established through Makefile variables such as "obj-y", which FEVER does not extract, had little influence on the accuracy of the mapping change extraction (with at least 98% accuracy for mapping changes). Such mapping appears to be more stable, and thus are less present in our data. Nonetheless, from an evolution point of view, FEVER performed as expected.

External validity. We devised our prototype to extract changes from a single large scale highly variable system, namely the Linux kernel. In that sense, our study is tied to the technologies that are used to implement this system:

the Kconfig language, the Makefile system and the usage of code macros to support fine-grained variability. However, there are several other systems using those very same technologies, such as aXTLs and uClibc, on which our prototype - and thus our approach - would be directly usable.

For other types of systems, one would have to adapt the model reconstruction phases depending on the system under study. If we consider another operation system such as eCos, one would have to rebuild the same change model from features described in the CDL language instead of KConfig. A similar work would be necessary to consider systems using the Gradle build system, rather than Makefile. However, the change model, based on an abstract representation of feature changes, should be sufficient to describe the evolution of highly variable systems, regardless of the implementation technology.

This work focuses on build-time variability, constructed around the build system and an annotative approach to fine-grained variability implementation (#ifdef statements). While we believe that the change model may be useful to describe runtime variability, the extraction process is not suitable to extract feature mapping from the implementation itself at this time. We cannot extend this work to runtime variability analysis without further study.

7. THE FEVER DATASET

In this section, we provide an overview of the feature evolution in the Linux kernel during release v3.13 captured by FEVER. Then, we present three practical scenarios where FEVER can be of use. Finally, we elaborate on further potential usage of the FEVER dataset.

7.1 Co-evolution in Linux

The feature-oriented co-evolution of artefacts has been studied in the past, as mentioned in Section 3. Previous studies describing complex changes relied on *manual* analysis of commits and do not provide a quantitative overview of how frequent co-evolution of artefacts is during feature evolution. With FEVER, this is possible. In this section, we rely on the FEVER data extract from release v3.13 of the Linux kernel.

Let us first consider the coverage of **TimeLine** entities in terms of commits. In release v3.13, we captured 13,288 commits. Among those, 11,859 (89.2%) are related to at least one **TimeLine** entity. Among the 1,429 commits that are not connected to a **TimeLine**, 1,209 relate to merge operations, tagged releases or other maintenance operations. The remaining commits affect files which are not source, build nor variability model related.

We focus on how features evolved in this release, and the spaces affected by their evolution. The number of **TimeLine** entities is the number of features that have seen their core behaviour or influence modified in the course of the release. We can then, for each of them, determine in which spaces this evolution took place.

The dataset contains 4,480 **TimeLine** entities. Among those **TimeLine** entities, 3,437 are connected to **commit** entities only through feature "core update" relationships. The majority (75.6%) of the features evolved due to changes to their declaration, mapping, or modifications to the files they are mapped to. Only 587 (13%) evolved only through "influence updates": their implementation did not change but they played a role in the evolution of the implementation of other features. The remaining 456 **TimeLine** entities have both "core -" and "influence update" relationships, indicating that for a minority of features, the evolution induced changes to both their implementation and their influences on other features.

We use the FEVER database to identify, for each of them, in which spaces the changes occurred. The resulting distribution is shown in Figure 5. The figure shows that most features evolved following a modification to their mapped source files (81%). Only 5% (243) of **TimeLine** entities exhibit changes in all three spaces.

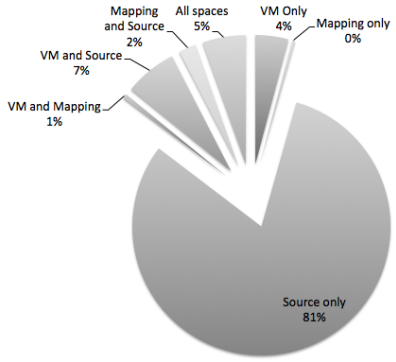

Figure 5: Spaces affected by feature evolution

A Linux release lasts six weeks, four of them are dedicated to bug fixes [31]. Most the development is then focused on fine-tuning the implementation of features. Moreover, new capabilities may also be supported by modifications of existing features. This would explain why most of the feature changes we observe are in the implementation. Nonetheless, for 19% of features, modifications to heterogeneous artefacts took place.

7.2 FEVER in Practice

The FEVER data is stored in a Neo4j graph database.[4] Every entity of the FEVER change meta-model is a node of the graph, and the relationships are edges. Data types are represented using node labels, and attributes are stored as node properties. The queries presented in this subsection are written in the Cypher query language.[5]

To illustrate the use of the FEVER dataset, let us consider the situation of a release manager building the release notes. He is interested in highlighting important features, and matching those to the commits that participated in their implementation. The release notes of Linux v3.13 [6] mention the following change "add[s] option to disable kernel compression" with a single commit. Looking at the commit, we know that a new configuration option named "KERNEL_UNCOMPRESSED" is introduced. We can check this with FEVER by querying the commits associated with the **TimeLine** of "KERNEL_UNCOMPRESSED" as follows:

```
match
(t:TimeLine)-[]->()<-[]-(c:commit)
where t.name = "KERNEL_UNCOMPRESSED"
return distinct c;
```

[4] http://neo4j.com/
[5] http://neo4j.com/docs/stable/cypher-query-lang.html
[6] http://kernelnewbies.org/Linux_3.13

This query returns two commits. The first, commit 69f055 mentioned in the release note is associated with a **FeatureEdit** entity denoting the addition of a feature. The second, commit 2d3c62 - occurring a few days later, is also associated with a **FeatureEdit** entity, but, surprisingly, *removes* the feature. A check in release v3.14 showed that the feature was never re-introduced. This means that the release notes written by the 3.14 release engineer were, in fact, incorrect. We argue that a dataset such as FEVER would have prevented this false entry in the release notes.

In another scenario, a developer is about to introduce a new driver for a touch-screen which should support the power management "SLEEP" feature. The developer might want to know how such support was done in other drivers. He queries the FEVER database for commits where a new feature (f1) is added (fe.change ="add"), and which interacts with a second feature (f2) whose name is "PM_SLEEP' as follows:

```
match (f1:TimeLine)-[:FEATURE_CORE_UPDATE]->
            (fe:FeatureEdit)<-[]-(c:commit),
 (c)-[]->()<-[:FEATURE_INFLUENCE_UPDATE]-(f2:TimeLine)
where f2.name = "PM_SLEEP"  and fe.change = "Add"
return f1,f2, distinct c;
```

In release v3.13 of the Linux kernel, this query returns ten results, giving the name of the newly introduced features, and the commits in which those changes occurred. Among the results, the developer might notice that feature "TOUCHSCREEN_ZFORCE" is among the results and might consider using this as an example to drive his own development.

A researcher in the domain of evolution of highly variable software systems might be interested in the typical structure of feature related changes. For instance, one might be interested in the introduction of abstract features, in the sense of Thuem et al. [32]: a feature only exists in the VM. We can identify the introduction of such features with this query:

```
match
 (t:TimeLine)-[:FEATURE_CORE_UPDATE]->(f:FeatureEdit)
where
 not (t)-[:FEATURE_CORE_UPDATE]->(:MappingEdit)
 and not (t)-[:FEATURE_CORE_UPDATE]->(:ArtefactEdit)
 and not (t)-[:FEATURE_INFLUENCE_UPDATE]->(:SourceEdit)
 and f.change="Add"
return t
```

In release v3.13, this query returns 42 features. Because **TimeLine** entities are regrouping changes across spaces and commits, we know that those 42 features are indeed abstract, and this is not the result of a developer who first modified the variability model and in a later commit adjusted the implementation.

One may be able retrieve similar information using a combination of Git and "grep" commands. We argue that obtaining the same information would require expert knowledge of features and their mapped artefacts, as well as a good knowledge of Git. With FEVER, a single query on the database is sufficient.

7.3 Further Applications

In a system such as Linux with 13,000 features, it might be difficult to pin-point which configurations should be used to test a new release, as testing all possible configurations is not feasible. The view of feature changes provided by FEVER provides additional information about commits, namely in terms of touched features. This information can be of use when deciding which configurations should be tested for defect following a code delivery.

The FEVER database could be combined with other existing data sources. Tian et al. devised a methodology to identify bug fixing commits in the Linux kernel [33]. Combined with the FEVER data, it is possible to identify the characteristics of changes leading to bug fixes, or find how features evolve during bug fix operations. This would in turn facilitate the work of Abal et al. to study the nature, the introduction and fixes to variability related bug [12].

The data provided by German et al. [31] can be used to track commits over time and across repositories. Combining this information with the FEVER database would allow us to track feature development across Git repositories, and observe how the Linux community collaboratively handles the development of inter-related features.

8. CONCLUSIONS

In this paper, we presented FEVER, an approach to automatically extract changes in commits affecting the implementation of features in highly variable systems. FEVER retrieves commits from versioning systems, and using model-based differencing, extracts detailed information on the changes, to finally combine them into feature-oriented changes. We applied this approach to the Linux kernel, and used the constructed dataset to evaluate its accuracy in terms of complex change representation. We showed that we were able to accurately extract and integrate changes from various artefacts in 82.6% of the studied commits.

Through this work, we make the following contributions. We first presented a model of feature-oriented changes, focusing on the co-evolution of feature representation in heterogeneous artefacts. We showed how we used model based differencing techniques to recover instances of the model from a SCM system in an automated fashion. We showed that the heuristics we used to obtain the change information yielded accurate results by applying the approach to the Linux kernel and manually validating the collected data. The collected data allowed us to show that co-evolution of artefacts during feature evolution does occur, but, over a single release, most features only evolve through their implementation. We presented practical scenarios in which FEVER can be useful for both developers and researchers. Finally, our prototype implementation and collected datasets are available for download.

The next step of our research is to establish a mapping between our change model and the co-evolution patterns as defined by Passos et al. [15] and the safe evolution templates proposed by Neves et al. [7]. We believe this might lead us to an automated identification of instances of known types of changes, and further identification of frequent complex changes in large scale systems. Furthermore, we will extended FEVER to more types of artefacts in order to apply this approach to a larger set of systems.

Acknowledgements

The authors thank Sven Apel for his feedback on the early versions of this work. 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.

9. REFERENCES

[1] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pp. 45–54, IEEE Comput. Soc, 2001.

[2] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake, "FAME-DBMS: tailor-made data management solutions for embedded systems," in *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, SETMDM '08, ACM Press, 2008.

[3] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise, "GENESIS: an extensible database management system," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1711–1730, Nov. 1988.

[4] I. Kumara, J. Han, A. Colman, T. Nguyen, and M. Kapuruge, "Sharing with a Difference: Realizing Service-Based SaaS Applications with Runtime Sharing and Variation in Dynamic Software Product Lines," in *Proceedings of the IEEE International Conference on Service Computing*, SCC '13, pp. 567–574, IEEE, June 2013.

[5] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *Proceedings of the international conference on Automated software engineering*, ASE '10, p. 73, ACM Press, 2010.

[6] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd International Conference on Software Engineering*, vol. 1 of *ICSE '10*, p. 105, ACM Press, 2010.

[7] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza, "Safe Evolution Templates for Software Product Lines," *Journal of Systems and Software*, 2015.

[8] 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, pp. 15–20, ACM, 2012.

[9] W. Heider, M. Vierhauser, D. Lettner, and P. Grunbacher, "A Case Study on the Evolution of a Component-based Product Line," in *Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA/ESCA'12, pp. 1 –10, 2012.

[10] 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, pp. 47–60, ACM, 2011.

[11] 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, pp. 107–116, 2012.

[12] 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, (New York, NY, USA), pp. 421–432, ACM, 2014.

[13] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in Software Evolution," in *International Workshop on Principles of Software Evolution*, pp. 13–22, IEEE, 2005.

[14] 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*, SPLC '14, (New York, NY, USA), pp. 274–283, ACM, 2014.

[15] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo, "Coevolution of variability models and related software artifacts," *Empirical Software Engineering*, pp. 1–50, May 2015.

[16] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial software systems: An empirical study," *Empirical Software Engineering*, Apr. 2015.

[17] 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, (New York, NY, USA), pp. 376–379, ACM, 2014.

[18] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, p. 1, ACM Press, 2013.

[19] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Computing Surveys*, vol. 47, pp. 1–45, June 2014.

[20] N. Dintzner, A. van Deursen, and M. Pinzger, "Analysing the Linux kernel feature model changes using FMDiff," *Software & Systems Modeling*, May 2015.

[21] 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, pp. 21–30, ACM, 2012.

[22] S. Zhou, J. Al-Kofahi, T. N. Nguyen, C. Kaestner, and S. Nadi, "Extracting Configuration Knowledge from Build Files with Symbolic Analysis," in *Proceedings of the 3rd International Workshop on Release Engineering (Releng)*, ACM Press, May 2015.

[23] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pp. 140–151, ACM Press, 2014.

[24] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, p. 81, ACM Press, 2013.

[25] 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, pp. 25–32, ACM Press, 2010.

[26] 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, pp. 81–86, 2009.

[27] A. Begel, K. Y. Phang, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 1 of *ICSE '10*, p. 125, ACM Press, 2010.

[28] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," tech. rep., Software Engineering Institute, Carnegie Mellon University, 1990.

[29] H. Spencer and G. Collyer, *#ifdef Considered Harmful, or Portability Experience With C News*. 1992.

[30] 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.

[31] D. M. German, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: an empirical study of how Linux uses Git," *Empirical Software Engineering*, Mar. 2015.

[32] 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, pp. 254–264, IEEE Computer Society, 2009.

[33] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux Bug Fixing Patches," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 386–396, IEEE Press, 2012.

TUD-SERG-2016-005
ISSN 1872-5392