A Dataset of Feature Additions and Feature Removals from the Linux Kernel

Leonardo Passos[*]
University of Waterloo
Canada
lpassos@gsd.uwaterloo.ca

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

ABSTRACT

This paper describes a dataset of feature additions and removals in the Linux kernel evolution history, spanning over seven years of kernel development. Features, in this context, denote configurable system options that users select when creating customized kernel images. The provided dataset is the largest corpus we are aware of capturing feature additions and removals, allowing researchers to assess the kernel evolution from a feature-oriented point-of-view. Furthermore, the dataset can be used to better understand how features evolve over time, and how different artifacts change as a result. One particular use of the dataset is to provide a real-world case to assess existing support for feature traceability and evolution. In this paper, we detail the dataset extraction process, the underlying database schema, and example queries. The dataset is directly available at our Bitbucket repository: `https://bitbucket.org/lpassos/kconfigdb`

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control

General Terms

Management

Keywords

Linux, Version Control History, Evolution, Traceability

1. INTRODUCTION

Highly-configurable software systems allow users to configure the target software according to their own preferences and needs. Configurability is achieved by having *variable software artifacts*, meaning that they can be restructured to suit a particular configuration [7]. Different software systems fit into such description (e.g., database management systems [3, 8, 11], SOA-based applications [2], and operating systems [1, 4, 5]), and the Linux kernel is probably the most well-known case.

In the Linux kernel, variability is captured in system features (configurable system options), which are explicitly declared in *variability models* written in the Kconfig language [9]. Features in variability models are then referenced in build files and C code. Figure 1 conceptually illustrates how features appear in these three artifact types and how they bind such artifacts. The variability model contains feature declarations, including drivers, file systems, scheduling policies, network protocols, etc. In the figure, feature FB is a driver providing support for framebuffer devices.[1] Through a configuration process over the declared features (step 1),[2] users state which features should be part of the final kernel, and which should be excluded. Upon a feature selection, specific build rules are triggered to compile corresponding C files. For example, selecting FB triggers the compilation of fb.c (step 2). Compilation, however, first requires pre-processing target source files to remove any compile-time variability introduced by C pre-processor directives (step 3), such as *ifdefs*. In our example, the pre-processing of sti_core.c, another framebuffer-related feature, shows that under the presence of FB, the post-processed sti_select_fbfont function has a non-NULL return. After pre-processing target files, the resulting directive-free files are compiled (step 4).

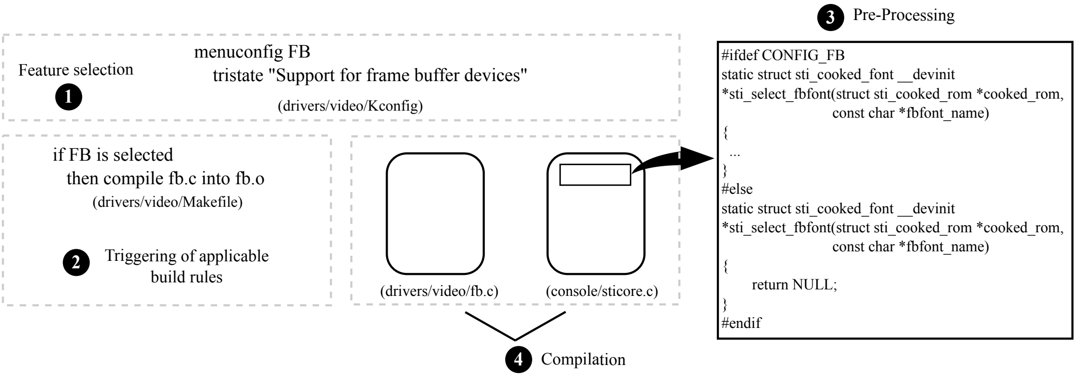
As the kernel evolves, new features are introduced, retired, split, merged and renamed. These evolution changes are expressed by two basic operations in the variability model: feature additions and feature removals (e.g., the split of a feature f into f_i and f_j is given by the removal of f followed by the addition of f_i and f_j). By processing the Linux kernel version control history, we extract a dataset of feature additions and removals to the variability model, linking them to their specific commits, changed files (Kconfig, build system, and C files), contributors and associated releases. Commits that do not change the variability model are also stored, but with less detail. The dataset, kept as a relational database, allows different types of queries, including the retrieval of the commit that adds/removes a particular feature, the release in which an addition/removal occurs, which contributors

[*] Funded by CAPES, grant BEX 0459-10-0.

MSR '14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05 ...$15.00.

[1] A framebuffer device is an abstraction for the graphic hardware. It represents the framebuffer of some video hardware, and allows application software to access the graphic hardware through a well-defined interface [6].

[2] Different tools exist to configure the kernel, including xconfig, menuconfig, and gconfig. These tools render the Kconfig model as a hierarchy of features, from which users select those of interest and set their values accordingly.



```
#ifdef CONFIG_FB
static struct sti_cooked_font __devinit
*sti_select_fbfont(struct sti_cooked_rom *cooked_rom,
                   const char *fbfont_name)
{
  ...
}
#else
static struct sti_cooked_font __devinit
*sti_select_fbfont(struct sti_cooked_rom *cooked_rom,
                   const char *fbfont_name)
{
        return NULL;
}
#endif
```

Figure 1: Example of variability in the Linux kernel

add more features, the places (directories) where most features belong, etc. Our dataset allows researchers to assess the evolution of the kernel from a feature-oriented point-of-view, in addition to providing concrete test cases for feature traceability and evolution techniques. The dataset contains over 300,000 commits, covering the kernel development from v2.6.12 (year 2005) to v3.9 (year 2013). In our own previous work [10], we have used a small subset (≈ 500 commits in the v2.6.26–v3.3 release range) of a simplified version of this dataset (not all tables and attributes were captured by the previous version) for collecting patterns of the coevolution of the Linux kernel variability model, Makefiles and C code.

In the following, we present the data extraction process (Sec. 2), the database schema of our dataset (Sec. 3), example queries (Sec. 4), and a discussion of current limitations (Sec. 5).

2. DATA EXTRACTION

To extract our dataset, we follow the process depicted in Fig. 2. The process starts by first collecting the release versions of the kernel by enumerating the release tags in the kernel repository. The collected tags are then ordered from oldest to newest, and output as ordered pairs (*release* i, *release* $i+1$). Step 1 in Fig. 2 comprises these two tasks. At step 2, we iterate over the ordered release pairs and inspect the commits between the releases of each pair, and at step 3 we parse their associated patches, extracting various data (see Sec. 3). After parsing (step 4), we store the collected data in a relational database, currently managed by PostgreSQL RDBMS.

The process in Fig. 2 is fully automated and has been implemented in a custom-made tool (kdb), whose source code is publicly available at `https://bitbucket.org/lpassos/kdb`. As the implementation of kdb requires connectivity with git, we have also implemented a thin IPC library (gitlib) to retrieve information from the Linux kernel git repository. Again, the library is released under GPL, and its source code can be downloaded from Bitbucket.[3] It is worth noting that the database does not store data relative to the content of patches changing non-Kconfig artifacts (e.g., Makefiles and C code). However, as we record all files changed in a given commit, any complementary information can be programmaticallly obtained with the use of gitlib.

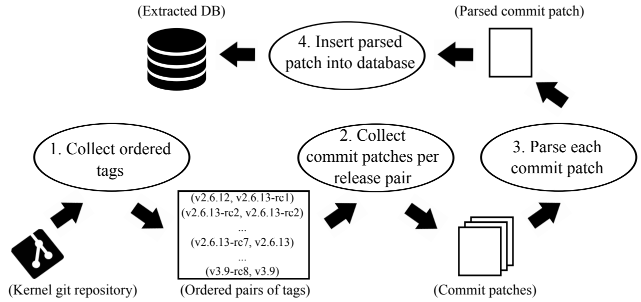


Figure 2: Data extraction process

3. SCHEMA

The database storing the collected data from each parsed commit follows the schema in Fig. 3. The database stores information relative to all commits, saving which files are modified, the release interval that a commit takes place, and who is the committer and who is the author of the change, as given by the corresponding tables in the schema. Changes (table file_change) are also specialized into file renames (in that case we also store the similarity degree (%) to the original file), file copies, file additions, file deletions and file modifications. In the case of feature changes in Kconfig files, we also record which features are added and which are deleted, as given in feature_change_unit. Thus, the type of a feature change, given in table feature_change_unit_type, is either an addition (id = 'A') or removal (id = 'D').

4. EXAMPLE QUERIES

In the following, we provide some example queries that can be answered by our database. Our examples are given in terms of a particular framebuffer-related feature: FB_IMAC.

EQ. 1) What is the commit that adds/removes a given feature?

This query applies when it is known that the kernel has removed a feature, but the associated commit is unknown. The SQL-select bellow retrieves such a commit:

```
select commit.hash      as commit_hash,
       file_change.file as file
from
  file_change, feature_change_unit, feature, commit
where
  file_change.fk_commit = commit.id and
  feature_change_unit.fk_file_change = file_change.id and
```

[3] `https://bitbucket.org/lpassos/gitlib`

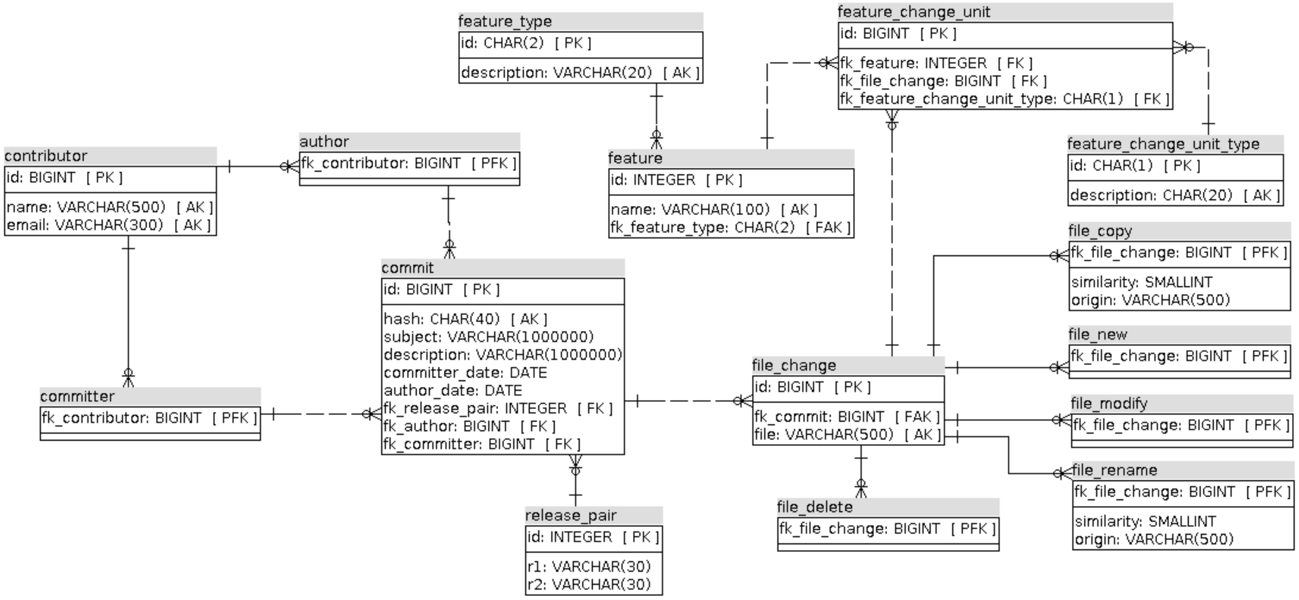

feature_type
id: CHAR(2) [PK]
description: VARCHAR(20) [AK]
feature_change_unit
id: BIGINT [PK]
fk_feature: INTEGER [FK]
fk_file_change: BIGINT [FK]
fk_feature_change_unit_type: CHAR(1) [FK]
contributor
id: BIGINT [PK]
name: VARCHAR(500) [AK]
email: VARCHAR(300) [AK]
author
fk_contributor: BIGINT [PFK]
feature
id: INTEGER [PK]
name: VARCHAR(100) [AK]
fk_feature_type: CHAR(2) [FAK]
feature_change_unit_type
id: CHAR(1) [PK]
description: CHAR(20) [AK]
file_copy
fk_file_change: BIGINT [PFK]
similarity: SMALLINT
origin: VARCHAR(500)
commit
id: BIGINT [PK]
hash: CHAR(40) [AK]
subject: VARCHAR(1000000)
description: VARCHAR(1000000)
committer_date: DATE
author_date: DATE
fk_release_pair: INTEGER [FK]
fk_author: BIGINT [FK]
fk_committer: BIGINT [FK]
file_new
fk_file_change: BIGINT [PFK]
committer
fk_contributor: BIGINT [PFK]
file_change
id: BIGINT [PK]
fk_commit: BIGINT [FAK]
file: VARCHAR(500) [AK]
file_modify
fk_file_change: BIGINT [PFK]
file_rename
fk_file_change: BIGINT [PFK]
similarity: SMALLINT
origin: VARCHAR(500)
file_delete
fk_file_change: BIGINT [PFK]
release_pair
id: INTEGER [PK]
r1: VARCHAR(30)
r2: VARCHAR(30)

Figure 3: Database schema

```
  feature_change_unit.fk_feature_change_unit_type = 'D'
  and feature_change_unit.fk_feature = feature.id and
  feature.name = 'FB_IMAC'
```

Running the query indicates that FB_IMAC is removed from drivers/video/Kconfig in commit 7c08c9ae0c145.

EQ. 2) At which release is a given feature added or removed?

This query builds on the previous one, and allows retrieving the latest release in which the feature still exists (from_release), and the immediate release in which it is no longer present (to_release):

```
select release_pair.r1 as from_release,
       release_pair.r2 as to_release
from
  file_change, feature_change_unit, feature, commit,
  release_pair
where
  commit.fk_release_pair = release_pair.id and
  file_change.fk_commit = commit.id and
  feature_change_unit.fk_file_change = file_change.id and
  feature_change_unit.fk_feature = feature.id and
  feature.name = 'FB_IMAC' and
  feature_change_unit.fk_feature_change_unit_type = 'D'
```

The query result indicates that release v2.6.27 is the last stable one to contain FB_IMAC, and the next immediate release (v2.6.28-rc1) no longer has it in the variability model.[4]

EQ. 3) What are candidate changes where feature additions/removals are likely to denote something else (e.g., a merge, a rename, a split, etc)?

This query allows identifying situations where feature additions and removals are composed to achieve more complex changes, such as the merge or split of features. There are many ways on how to write such a query. As a heuristic-based criteria for finding merges, stated in the SQL-select in the following, we filter commits that necessarily remove a single feature (lines 17–22), but do not add any new ones (lines 22–28). Our strategy is to search for features that are removed from the variability model, but are merged with existing features by changes in other artifacts. Hence, we state that target commits must modify at least an implementation file (.c), a header file (.h), or a Makefile (lines 10–17):

```
1  select commit.hash  as commit_hash,
2         feature.name as feature_name
3  from
4    commit, file_change, feature_change_unit, feature
5  where
6    file_change.fk_commit = commit.id and
7    feature_change_unit.fk_file_change = file_change.id
8    and feature_change_unit.fk_feature = feature.id and
9    feature_change_unit.fk_feature_change_unit_type = 'D'
10   and (select count(*)
11    from   file_change as fc, file_modify as fm
12    where  (fc.file like '%.c' or
13            fc.file like '%.h' or
14            fc.file like '%Makefile') and
15            fc.fk_commit = commit.id and
16            fm.fk_file_change = fc.id
17   ) > 0 and (select count(*)
18    from file_change as fc, feature_change_unit as fcu
19    where fc.fk_commit = commit.id and
20          fcu.fk_file_change = fc.id and
21          fcu.fk_feature_change_unit_type = 'D'
22   ) = 1 and (select count(*)
23    from file_change as fc,
24         feature_change_unit as fcu
25    where fc.fk_commit = commit.id and
26          fcu.fk_file_change = fc.id and
27          fcu.fk_feature_change_unit_type = 'A'
28   ) = 0
```

One of the commit hashes returned by this query lists the commit hash of the commit removing FB_IMAC. The inspection of that commit shows that, while FB_IMAC is removed from the variability model, part of its implementation is copied to another existing feature (FB_EFI), characterizing a merge between the two (for a full discussion of this case, the reader is referred to [10]). In this case, the functionality provided by FB_IMAC is still supported, although now in a different feature. Thus, running such a query allows us to

[4] In the Linux kernel, unstable releases are suffixed with rc, whereas stable ones are not.

provide concrete cases for assessing techniques supporting feature evolution and traceability.

For example, techniques that reason on the evolution of the variability model only (e.g., [12]) would not be able to capture this case adequately, since removal of FB_IMAC from the variability model does not imply that support for FB_IMAC has been dropped.

Other Queries. Examples of other queries include:

- *Which contributors add more features in the kernel?* This query can be answered by looking the contributors' records associated with file changes that necessarily add features, provided that the same features are not excluded in the same commit (e.g., if features are relocated in their Kconfig file, the database keeps two records in the file_change table; one for removing the feature declaration, and another for adding it in a different place in the same file).

- *Where are most of the features located in the kernel?* As the database keeps the location of each feature that is added, we can group feature additions according to the top-level directory where their Kconfig files belong (e.g., inside the drivers folder of the kernel source code tree). This can point to places where the kernel contains most of its features.

- *What is the average time for staging*[5] *features to be merged into the main kernel, or removed from it?* To write such a query, one must filter commits whose files belong to the staging subfolder inside drivers, as recorded in the path of changed files (column file, table file_change).

- *To what degree changes in Kconfig files are accompanied by changes in other artifacts?*

Due to limited space, we omit the SQL-select statement of these queries.

5. DATASET LIMITATIONS

Our dataset has some limitations. Although it stores all commits, the database does not hold all patch changes, as we do not parse the addition/removal of functions in C code, build rules in Makefiles, etc. Rather, we restrict to parsing Kconfig files that necessarily add/remove features, and store associated data. In the case of changes in other artifacts, we only save which files are changed, and whether they are copied, renamed, added from scratch, deleted, or had their content modified.

Moreover, we do not automatically identify situations in which the addition or removal of features in the variability model indicate a richer change semantics, such as the merge of features or a split situation. This limitation can be overcome if ones uses the database, together with extra queries directly issued to git. In such cases, gitlib can be used. The dataset, as is, already allows researchers to query specific situations to evidence such cases, but requires manual analysis to discard false-positives.

Another limitation of our dataset is that it concerns the evolution of a single system: the Linux kernel. We aim to overcome this limitation in the near future, as our custom-made tool (kdb) can already be applied to different systems, provided they rely on the Kconfig language for encoding variability models and use git as their version control system. Other variant-rich software systems satisfy such criterion, specially in the systems software domain [5].

6. REFERENCES

[1] S. Apel, D. Batory, C. Kstner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.

[2] L. Baresi, S. Guinea, and L. Pasquale. Service-Oriented Dynamic Software Product Lines. *Computer*, 45(10):42–48, 2012.

[3] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988.

[4] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski. Feature-to-code Mapping in Two Large Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, pages 498–499. Springer, 2010.

[5] 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, 2013.

[6] A. Buell. Framebuffer HOWTO. `www.tldp.org/HOWTO/Framebuffer-HOWTO/`, 2008. Last seen: February 16th, 2014.

[7] J. V. 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*, pages 45–54. IEEE, 2001.

[8] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th International Software Product Line Conference*, pages 223–232. IEEE, 2007.

[9] Kconfig. The Kernel Configuration Language (Kconfig). `www.kernel.org/doc/Documentation/kbuild/`, 2008. Last seen: February 16th, 2014.

[10] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proceedings of the 17th International Software Product Line Conference*, pages 91–100. ACM, 2013.

[11] 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*, pages 1–6. ACM, 2008.

[12] T. Thüm, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 254–264. IEEE, 2009.

[5] Staging features in the kernel are features that are not fully functional, or that are not mature for official inclusion. These features are contained in the kernel to allow users to have early support for a given functionality (e.g., a driver).