Feature-Oriented Evolution of Variant-rich software systems

Dintzner, Nicolas

DOI
10.4233/uuid:d23770ce-51ad-43d3-960b-3fa2ad7623f1

Publication date
2017

Document Version
Final published version

Citation (APA)
Dintzner, N. (2017). *Feature-Oriented Evolution of Variant-rich software systems*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:d23770ce-51ad-43d3-960b-3fa2ad7623f1

FEATURE-ORIENTED EVOLUTION OF VARIANT-RICH SOFTWARE SYSTEMS

FEATURE-ORIENTED EVOLUTION OF VARIANT-RICH SOFTWARE SYSTEMS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 7 juli 2017 om 10:00 uur

door

Nicolas Joseph René DINTZNER

Ingénieur diplomé
E. P. F. Ecole d'ingénieurs, Sceaux, France
geboren te Parijs, Frankrijk.

Dit proefschrift is goedgekeurd door de

promotor: Prof.dr. A. van Deursen
promotor: Prof.dr. M. Pinzger

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. A. van Deursen	Technische Universiteit Delft
Prof. dr. M. Pinzger	University of Klagenfurt
Onafhankelijke leden:	
Prof.dr. K.G. Langendoen	Technische Universiteit Delft
Prof.dr.ir. D.H.J. Epema	Technische Universiteit Delft
Prof.dr. J.J.M. Hooman	Radboud University Nijmegen
Prof.dr.ir. S. Apel	University of Passau
Prof.dr. A. Wąsowski	IT University of Copenhagen

This research was supported by the Dutch national program COMMIT and carried out as part of the Allegio project.

Keywords: Variabitlity, feature, system, evolution

Printed by: Sieca Repro

Front & Back: The illustration of the cover is based on a photographic portrait of Charles Darwin, published by John G. Murdoch. The image itself was obtained from Wikipedia.org *https://en.wikipedia.org/wiki/Portraits_of_Charles_Darwin.*

 ISBN 978-94-6186-829-9

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

This work is dedicated to my familly.

ACKNOWLEDGEMENTS

This thesis is the conclusion of almost five years of research work. While a Ph.D. research is generally considered as personal endeavour, a number of people made this work possible, either directly or indirectly.

First and foremost, I extend my most sincere gratitude to my two promotors: Martin Pinzger and Arie van Deursen. I would like to thank Martin for his expertise and guidance through all my research activities. His feedback on the different aspects of my work, from the big abstract questions down to minute typos, was invaluable and made this thesis possible. I am also very grateful to Arie for his unwavering support and mentorship throughout those five years of work. Through his guidance, Arie helped me put the "philosophy" in my "Ph.D.". I thank both of you for your unfaltering guidance and enthousiastic support during those 5 years.

I would also like to thank my reading committee, for their time and valuable feedback: Prof. Jozef Hooman, Prof. Sven Apel, Prof. Andrzej Wąsowski, Prof. Dick Epema and Prof. Koen Langendoen. Their insightful comments and questions allowed me to perceive my own work on a wider perspective.

I performed my research work in cooperation with TNO-ESI and Philips Healthcare. In this context, I met David Watts. During our collaboration, he provided support and regular feedback, facilitated my work with Philips, and did the heavy lifting with respect to our partner. His help was everytime much welcome and very appreciated. I would like to extend a big thanks to the Ph.D. students I worked with while staying at Philips: Steven, Freek, and Sarmen.

Throughout my Ph.D. journey, I meet a number of researchers whose influence on me was more than positive, as a researcher and a person. I was lucky enough to work with Uira Kulesza, who showed me a relaxed way of doing very serious work. Leonardo Passos and Leopoldo Teixeira, with whom I spent hours looking at Linux commits, helped me create the backbone of this thesis. Finally, Sandro Schulze and Gilles Perrouin both taught me how to do research with a beer, or wine and how to chose between the two.

Aldo, Dena, Olga, and Natalia are the people who I grew up with during my Ph.D. Thanks to them, I got to see how research was performed in fields other than mine, how the challenges were different yet similar. It was a pleasure to share drinks with them when celebrating accepted and rejected papers alike, and mild hangovers that followed.

I must thank as well all the members of the Software Engineering Research Group. Thank you for the feedback on papers and presentations, thank you for helping me with small technical difficulties and hard questions. It was a pleasure to work with you all.

Then, I should give credit to some people I met during my PhD, who, from some reasons, made my PhD years sweeter by just being around. Thank you Boris, Delphine, Kishor, Oana.

I have to thank my personal support team: Nicolas Beziere, Nicolas Lefebvre, Pierre Gaudemer-Houzard, Marie-Emmanuelle Ricour, and Regis Behmo. It is a privilege to have you as friends.

I conclude this section by thanking my family who made all of this possible. Monique, Anne, Lorraine, and Jean-Pierre. Thank you.

CONTENTS

SUMMARY

Most modern software systems can be adjusted to satisfy sets of conflicting requirements issued by different groups of users, based on their intended usage or execution context. For systems where configurations are a core concern, specific implementation mechanisms are put in place to allow the instantiation of sets of tailored components. Among those, we find selection processes for code artefacts, variability-related annotation in the code, variability models representing the available features and their allowed combinations.

In such systems, features, or units of variability, are scattered across the aforementioned types of artefacts. Maintenance and enhancement of existing systems remain a challenge today, for all types of software systems. But in the case of variant-rich systems, engineers face an additional challenge due to the complexity of the product instantiation mechanisms: the maintenance of the variability model of the system, the complex build mechanisms, and fine-grained variability in the source code. The evolution of the system should be performed such that the information contained within the various artefacts remains consistent. In practice, this means that as the implementation of the system evolves, so should the mechanisms put in place to generate tailored products.

Little information is available regarding changes occurring in such systems. To efficiently support such developers tasks and ease maintenance and enhancements activities, we need a deep understanding of the changes that take place in such systems. The state of the art provides trends over long period of times, highlighting systems growth - such as number of added or removed features in each release, or the evolution of cross-tree constraints in a variability model. While important to describe the core dynamics behind the evolution of a system, this does not provide information on the changes performed by developers leading to such trends. Similarly, this global information cannot be leveraged to facilitate developers' activities.

The focus of this thesis is the acquisition and usage of change information regarding variant-rich system evolution. We show how the information lacking from today's state-of-the-art can be obtained from variant-rich system change history. We propose a set of tool-supported approaches designed to gather such information and show how we leverage change information for change impact analysis, or to derive knowledge on developer practices and the challenges they face during such operations. With this work, we shed new light on change scenarios involving heterogeneous artefacts regarding their nature as well as their prevalence in the evolution of such complex systems, and change impact analysis in variant-rich systems.

We designed a model-based approach to extract feature related changes in heterogeneous artefacts. With such an approach, we can gather detailed information of feature evolution in all relevant artefacts. We created an approach for multi-product line modeling for impact computation. We leverage variability information to produce a collection of inter-related variability models, and show how to use it for targeted feature-

change impact analysis on available capabilities of the product family. By applying our change extraction approaches on the Linux kernel, we were able to empirically characterize the evolution of the variability model of this system. We showed that the variability model of that system evolves mostly through modification of existing features, rather than through additions and removals. Similarly, we studied co-evolution of artefacts during feature evolution in the Linux kernel. Our study revealed that, in this system, most features evolve mostly through their implementation, and complex changes, involving heterogeneous artefacts are not the most frequent.

Through this work, we provide detailed information on the evolution of a system, namely the Linux kernel, and the means used to obtain this information. We show that the gathered data allow us to reflect on the evolution of such a system, and we argue that gathering such information on any system is a source of valuable information regarding a system architecture. To this end, all tools developed in the context of this study were made available to the public.

In this work, we provide key information on the evolution of the Linux kernel, as well as the means to obtain the same information from other variant-rich systems. The knowledge gained on common evolution scenarios is critical for tool developers focusing on the support of development of variant-rich systems. A better understanding of common evolution scenario also allows engineers to design systems that will be better equipped to elegantly evolve through such scenarios. While a number of challenges will still have to be addressed in this domain, this work constitutes a step toward a better understanding of variant-rich system evolution and therefore toward better variant-rich system designs.

1

INTRODUCTION

Everything starts somewhere, though many physicists disagree. But people have always been dimly aware of the problem with the start of things. They wonder how the snowplough driver gets to work, or how the makers of dictionaries look up the spelling of words.

Terry Pratchett, *The Hogfather*

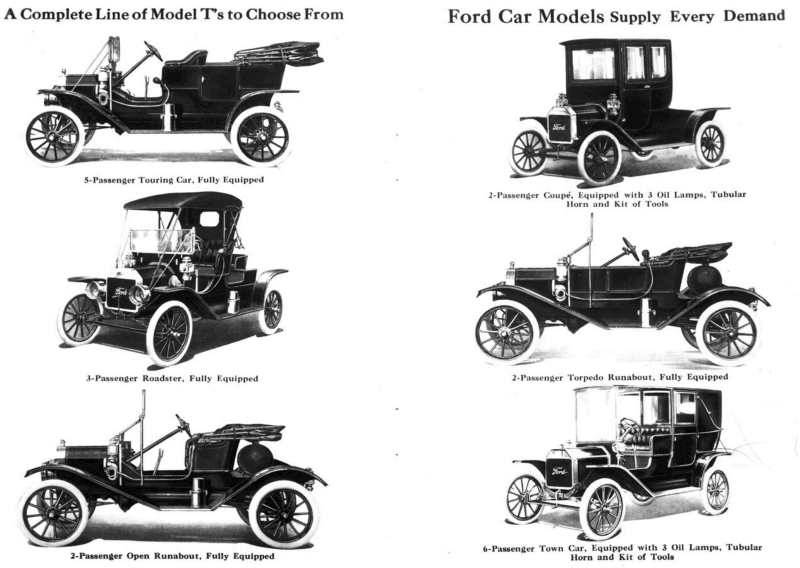

Figure 1.1: Variants of the Ford T model

1. VARIANT-RICH SYSTEMS

Software systems designed for a large group of users are often configurable. This need for configurable systems comes from the variations between user requirements within the targeted user groups. Such systems can be adjusted, or tuned, to match as closely as possible user requirements. Differences between variants of the system may stem from their supported execution environments, for systems running on various platforms, their different functional requirements, or may be the result of market segmentation, regulatory constraints, and so on.

1.1. VARIANT-RICH SYSTEMS IN THE WILD

Providing tailored products for specific usages is a common practice and has been used in various industrial domains for a very long time. For instance, the automotive industries started to offer a wide range of very similar cars quite early on in its history. As an example from this domain, the different versions of the early Ford T models are depicted in Figure 1.1. In this picture, we can see the same model ("T") and its variants, each designed for a specific context.

Nowadays, the video game industry produces the same game on multiple platforms. In Figure 1.2, we see three variants of the video game "Battlefield 1". While offering very similar gaming experience, each variant is designed to run on specific hardware, and be used with specific gaming interfaces (keyboard and mouse, or gamepad).

Differences between variants may be more significant than simply runtime environ-

Figure 1.2: Variants of the "Battlefield 1" video game (Courtesy of *EA Games*)

ments. For instance, Oracle offers several variants of the MySQL database system[1]. Each variant comes with a different set of capabilities. The more capabilities contained within the variant the more expensive it is to acquire it.

Finally, in the domain of healthcare equipment, Philips Healthcare develops highly configurable medical equipment such as the Allura X-ray system. The system itself, designed for interventional x-ray imaging, comes with a number of configuration options, and should interface with existing systems within the host hospital. Each variant may differ by its hardware (table, mechanical arms...), its software (imaging capabilities), its interfaces with external equipment, and its interfaces with the host hospital information system (to retrieve patient data for instance). Once the system is fully configured, and installed, the system is virtually unique.

More generally, we can observe such configurability of systems in a large range of domains: distributed systems (servers, SaaS, SoA) (Juanjuan Jiang et al., 2005; Kumara et al., 2013; Mietzner et al., 2009; Queiroz et al., 2014), database management systems (Queiroz et al., 2014; Siegmund et al., 2013), operating systems (Dintzner et al., 2015a; Hubaux et al., 2012; Sincero et al., 2010a), health care systems (Dintzner et al., 2015b; Holl et al., 2012), high performance computing (Lengauer et al., 2014), and a number of industrial domains.[2]

While closely related, each variant of the system is different from others. Strictly speaking, each variant is a different system. Software engineering practices provide approaches and techniques to design and implement single systems. From the requirement gathering phases to validation tests, we have today very clear methods to develop high-quality products. However, when the number of variants is high, standard development practices cannot be applied. Applying single product approaches to each variant, to ensure the highest possible quality of each of them, would be too time and resource consuming. Moreover, because of the overlap between variants, most of the effort would be wasted by repeating the same design and implementation steps for each variant. Therefore, applying "single-product" approaches for each variant is not a practical way to build variant-rich systems.

Efficient re-use of development artefacts between variants is considered as a key factor to the successful and efficient development of groups of related software systems. In

[1] https://www.mysql.com/products/
[2] http://splc.net/fame.html

1

1976, David Parnas suggested that, when developing program families - or groups of programs sharing a number of common characteristics, re-use would be better achieved if common characteristics of the members were considered together at first as a whole, before focusing on their differences (Parnas, 1976). By doing so, Parnas suggests that re-use of the artefacts built to support the common characteristics would be easier to re-use in all members of the program family. Re-use can of course be achieved at an implementation level, i.e., re-use of routine and methods through the re-use of code libraries, but also at a design level i.e., re-use of components and design elements. While opportunistic re-use can hasten the development process, systematic and organized re-use seemed to be a better approach to develop groups of similar systems.

Development approaches considering the family "first", rather than individual members are expected to yield a number of benefits (Clements and Northorp, 2002; Lemley and O'Brien, 1997; Lim, 1994). First, since the different products are created from a pre-existing set of software components, the development of new family members should be cheaper. A number of components do not need to be redesigned and re-implemented, therefore one saves effort and time. Secondly, re-use should help in increasing product quality since each component has been used and tested over a longer period of time.

However, efficient re-use of all development artefacts, namely requirements, specification, software components, or functions is no trivial feat. Parnas (Parnas, 1976) suggests a generic re-use approach at a very abstract level. Once we put such an approach in practice, we see that the sheer number of artefacts and artefact relationships to be taken into account grows fast as the system under design grows more complex. More advanced methodologies such as product line engineering propose a more concrete framework to organize the development process and the resulting artefacts. Product line engineering approaches have been successful in a number of domains but are known to be hard to put in place, and they imply a very specific work organization such as the separation of the development of re-usable components and the development of specific variants.

This being said, most modern systems come with a number of variants, that need to be managed by development teams, and are not developed using strict product line engineering principles. For instance, research on product line practices adoption shows that companies and development teams developed variant-rich systems without applying the product line engineering practices (Clements et al., 2006; Koziolek et al., 2016; Li et al., 2016). Similar observations can be made regarding research on "clone and own" approaches (Fogdal et al., 2016; Rabiser et al., 2016).

1.2. SCOPE OF THIS THESIS

In the context of this thesis, we focus on systems from which variants are derived from a single set of artefacts: developers work on a set of files, and from this unique set, a number of variants can be produced. Regardless of the development practices used for the realization of such systems, they all are considered "variant-rich", i.e., from one design, one group of assets, we can derive a number of different products. The variants are created by composing elements contained within a given repository. In such contexts, the repository contains information on all variants, their allowed configurations and the mechanisms to be used to derive a concrete and valid variant from the available arte-

facts.

The main challenge that arises from such very common situations is that when a developer changes a single artefact, regardless of the type of that artefact, (s)he has to take into account all the variants of the system that the change might affect. Moreover, if the change is significant, (s)he might have to consider how to modify related artefacts to guarantee the consistency of the modification in the behaviour of the system, its possible configurations, and its impact on the product derivation mechanisms in place.

This makes the development, maintenance, and evolution more complex than for single variant systems. All development challenges regarding architecture, component coupling, and code complexity still need to be addressed as those systems are complex software systems. However, instead of considering changes within a single system, developers need to constantly think of any modification they perform as modifications of many different systems.

2. ANATOMY OF VARIANT-RICH SYSTEMS

In this section, we present the key artefacts that developers work with in the context of variant-rich system development. For each artefact, we present its role and its possible format, as well as its relationships with other artefacts. An understanding of those types of artefacts and their relationships is necessary to understand the complexity of the evolution process. While we expand on this idea throughout this thesis, we can already point out the following: a change performed by a developer on one of the artefacts presented below should be consistent with all other artefacts, or should impact all other artefacts consistently. This is the constraint imposed by the development of variant-rich systems using a single repository.

VARIABILITY SPECIFICATION

The members of a program family differ from each other in terms of features. In this thesis, we use the definition of *feature* suggested by Czarnecki et al. (2002): a **feature** is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family.

When working on a highly configurable system, the features of interest are the ones describing commonalities and variabilities of the different variants. Since those features are identified by actors of the software development process with very different roles and perspectives, we can expect features to be of various granularity and nature. The different examples provided by Kang et al. 1990 show that features can represent something as precise as a "move icon" capability for a software application, or something as encompassing as "horsepower" in the context of car design. Nonetheless, in both cases, those features are well-identified concepts that stakeholders can refer to when communicating about the system under study (Berger et al., 2015).

For projects with a relatively small number of features, we can find descriptions of features in configuration files. Such files describe the features and their values in a specific format which can be used as an input for the compilation of the system, or used as a reference while the system is running (runtime variability support). Figure 1.3 shows two examples of configuration files used in Torque3D,[3] a video game engine platform, and

[3] http://torque3d.org/

```
[main]
certname = puppetmaster01.example.com
server = puppet
environment = production
runinterval = 1h
strict_variables = true
```

System: Puppet - Language: Puppet configuration

```
// Set this to true to enable the ExtendedMove class.  This
// allows the passing of absolute position and rotation input
// device information from the client to the server.
$TORQUE_EXTENDED_MOVE = false;
```

System: Torque3D - Language: Torque configuration

Figure 1.3: Formalized configuration options - example from Torque3D and Puppet

Puppet,[4] a platform for build automation. We can see that, in both systems, options are identified by their name, and are associated with a value. That value may be a string, a numerical value, or a Boolean flag (such as"true" or "false"). In such a simple formalism, one cannot express the constraints between those features or their values.

Using a simple formalism is usually enough for projects with little variability. As a system grows, its features might start to depend on one another under certain circumstances and their allowed combinations (configuration) become harder to describe. Kang et al. proposed a formalism to describe features and their constraints (Kang et al., 1990). An example of this notation is depicted in Figure 1.4. It represents the variability of the BerkleyDB database system. The FODA notation allows the formalization of optional or mandatory features, selection of a single feature among a set, selection of at least one feature among a set, and finally, any Boolean expression of features as additional constraints (not represented in the graph). In this representation, features are identified by their names, and do not bear any attribute.

Many formalisms have been proposed to describe the variability of systems (Berger et al., 2013b; Kang et al., 1990; Schobbens et al., 2006). They differ by the details that can be added to a feature's description, and how to express composition rules. Figure 1.5 presents two configuration options, one described in the CDL language used in the context of the eCos operating system, and a second one described in the Kconfig language, used in the context of the Linux kernel. In Figure 1.5, we can see that the definitions of features are associated with more complex relationships, describing composition rules.

For such systems, the list of features and their composition rules constitute the *variability model* of the system: it is a formal description of the variability of the system under study. For instance, in the simple feature model proposed by Kang et al. in (Kang et al., 1990), composition rules, named "cross tree constraints", are Boolean expressions composed of feature terms. They are not part of the definitions of any specific features, but are part of the diagram. A different approach is taken for the Kconfig language, or the eCos CDL language, where the composition rules are expressed as part of the definition of the feature, as shown in Figure 1.5, where feature GENERIC_IOMAP "depends" on the satisfaction of the expression "HAVE_GENERIC_IOMAP && FOO" (where both

[4] https://puppet.com/

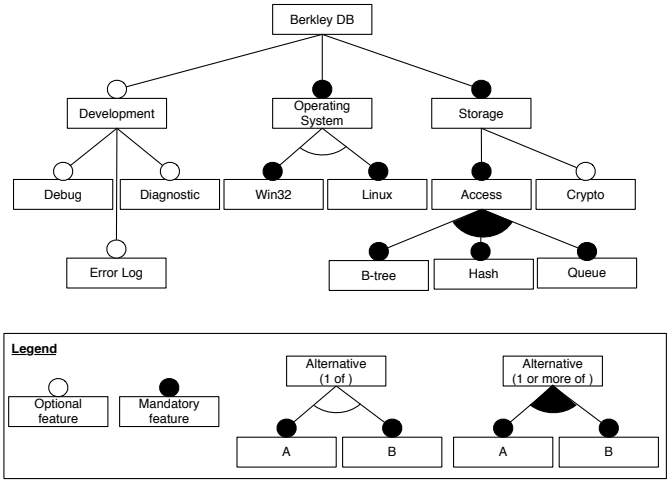

Figure 1.4: Partial feature model of the BerkleyDB database system, represented using the FODA notation. Adapted from (Rosenmüller et al., 2008)

```
cdl_package CYGPKG_INFRA {
    display       "Infrastructure"
    default_value 1
    include_dir   cyg/infra
    requires 1 == CYGINT_KERNEL_SCHEDULER
    description   "
        Common types and useful macros.
        Tracing and assertion facilities.
        Package startup options."

    compile startup.cxx prestart.cxx pkgstart.cxx userstart.cxx \
            dummyxxmain.cxx memcpy.c memset.c delete.cxx \
            diag.cxx tcdiag.cxx
}
```

System: eCos - Language: CDL

```
config GENERIC_IOMAP
     depends on HAVE_GENERIC_IOMAP && FOO
```

System: Linux Kernel - Language: Kconfig

Figure 1.5: Formalized configuration options - examples from eCos and the Linux kernel

HAVE_GENERIC_IOMAP and FOO refer to feature names). A valid configuration of the system is one in which if the feature "GENERIC_IOMAP" is present, then both features "'HAVE_GENERIC_IOMAP" **and** "FOO" are also present.

Such models serve two main purposes. First, they act as a specification of the system. If the variability model specifies that two features may be present together in variants, then their associated behaviour should be consistent with the system's specification. Secondly, variability models are used as a source of information for tools facilitating the creation of valid configurations - i.e., configurators (Krueger, 2006; Sincero and Schröder-Preikschat, 2008). Those purposes should be fulfilled by any variability models, regardless of the notation used.

If the number of features and constraints is large, users will have difficulties to create a selection of features that will satisfy all the composition rules (the Linux kernel has more than 13,000 features, and on average 2.5 constrains per feature (Lotufo et al., 2010)). In this context, a *configurator* is a tool helping users to create a valid configuration based on feature definitions contained in the variability model of the system. It usually presents the list of available features in a hierarchical form. As the user selects features to include in the configuration, the *configurator* may check whether the current selection is valid or not, and update the visible features that can be selected and added to the current configuration. For such tools, a feature is "selectable" if, given the set of already selected features, the feature can be included in the current configuration without violating any constraints.

From such a representation, we can determine the "presence condition" of a feature in a variability model. The "presence condition" of a given feature is a Boolean expression of features which, if satisfied, allows the selection of that feature.

ASSOCIATING BEHAVIOUR TO FEATURES

Given a valid configuration of the system, a user can expect the system to display all the characteristics, both functional and non-functional, specified by the selected features. As mentioned in the previous section, the selected features can have very different granularity, and their associated behaviour can therefore be supported in a very small code fragment, or in complete sets of components. In any case, features are associated with concrete artefacts that should be combined to create the requested variants.

The artefacts or concepts associated with a feature may vary. For instance, one may describe the requirements of the systems in terms of features, and start specifying the behaviour on a feature basis as well (feature-oriented design). In such a situation, a feature may be associated with formal models, or model fragments, describing the expected behaviour or properties of the system if those features are selected. Examples of models or fragments of models associated with features could be classes in a class diagram (Seidl et al., 2012), elements in a UML model (Czarnecki and Antkiewicz, 2005), or transitions and states (Classen et al., 2013). We note that in such examples, previously existing models are enhanced with additional feature-related information. Such approaches may be used to specify the behaviour of the system, depending on the included features, as well as perform formal verification on the systems and its variants as suggested by Classen et al. (Classen et al., 2013).

Features may also be associated with behaviour described in source code, i.e., concrete implementation artefacts. This association may be done in number of ways. It will

depend on the binding time of the feature, and the implementation strategy used may vary. For variation resolved at build time, the association may be performed in the build script (such as Makefile, in the context of the Linux kernel). For variation points resolved at runtime, the value of a feature may be checked in a simple "if" statement. In such cases, the code fragment associated with the feature is the code contained within the "if" statement.

For variation points resolved at build time, Figure 1.5 shows how, in the CDL language, a feature can be associated with a number of implementation files (with extension ".cxx"). In the context of the Linux kernel, the association - or mapping, between a feature and its concrete implementation is performed in the build system (comprised of Makefiles) and is expressed as follows:

```
obj-$(CONFIG_GENERIC_IOMAP) += iomap.o
```

In other systems, the association between a feature and its source may be implicit. In QT, a popular GUI framework, each feature is declared as a "package". While the terminology is different, the "packages" are identified during the installation process and the implementation located in the same directory as the package descriptor (package.xml file) is associated with the features. Additionally, QT offers the possibility to declare dependencies or additional resources inside the package.xml file. [5] In that sense, the QT model, quite like in the eCOS CDL language, puts in a single format both the features and the pointers to their implementation artefacts.

The association between features and implementation artefact may be complex, but plays an important role in the design and maintenance of variant-rich systems (Neves et al., 2015; Passos et al., 2015). It remains a key technique to support build-time variability resolution for coarse-grained variability.

During maintenance, the association of features and assets should be updated when necessary. This operation is not in itself very different in single variant systems than in many-variant systems. When a developer adds an implementation artefact, he should update the Makefile - very much like in any other system. Failure to do so guarantees that the artefact will not be taken into account during the compilation process. However, in the case of a single variant system, the Makefile content will be "simpler" - in the sense that it will refer to fewer external variables, and the presence conditions associated of a file in the list of artefacts to compile will be simpler. Variability does not change the nature of development activities related to the build system but increases the complexity of such artefacts.

FEATURE BEHAVIOUR IMPLEMENTATION

To support variability implementation at a more fine-grained level, implementation fragments may also be associated with specific sets of features. Regardless of the approach taken to associate code elements to features, the objective remains the same: given a valid configuration, one should be able to identify implementation artefacts, and implementation artefact fragments necessary to support the behaviour of the features of the selected configuration.

[5] http://doc.qt.io/qtinstallerframework/ifw-component-description.html

Compositional approaches advocate feature-based, or aspect-based decomposition of implementation artefacts. Given a valid configuration, those artefacts are composed, i.e., assembled in a prescribed way, to produce the final artefacts. Feature-oriented (Batory, 2004; Prehofer, 1997) and aspect-oriented programming (Kiczales et al., 1997) are examples of such implementation techniques. The composition mechanism may be as simple as selecting a file, or may involve more complex code generation tasks.

Annotative approaches take a different angle on the problem. The implementation fragments are annotated using feature names, or Boolean expressions of features. Upon product instantiation, the list of selected features is used to identify both the relevant artefacts and artefact fragments. The repository containing the implementation of the system contains all of the source code, all artefacts, but only a subset is used based on the set of selected features.

We illustrate annotative approaches with the code fragment below.

```
static inline int pfn_to_nid(unsigned long pfn){
  #ifdef CONFIG_NUMA
     return((int) physnode_map[(pfn) / PAGES_PER_ELEMENT]);
  #else
    return 0;
  #endif
  }
```

In this example, the method “pfn_to_nid” has two variants. If the feature NUMA is selected, the resulting pre-processed code will contain a variant of the method returning an element of the physnode_map array. If the feature is not selected, the pre-processed code of method pfn_to_nid will return 0, regardless of the input. This shows how, using annotations (#ifdef in this case), one can create a method with different compiled behaviour depending on the selected features. In large projects relying on annotative approaches, the implementation of a single feature is unlikely to be contained within a single artefact. Such features represent cross-cutting concerns (Bruntink et al., 2007). This leads to difficulties during maintenance operations as a developer needs to identify which artefacts are implementing a feature to fix or adjust its behaviour (Dit et al., 2013; Eisenbarth et al., 2003).

MAINTENANCE AND EVOLUTION

Maintenance and evolution of large-scale and complex systems are challenging (Godfrey and German, 2008; Lehman, 1996; Mens et al., 2005; Siy and Perry, 1998). In the context of highly variable systems, engineers face all the challenges that one might face when evolving a large scale system, and in addition, the variability and its implementation creates an additional degree of complexity. As the capabilities of the system evolve, the various artefacts participating in its implementation should evolve as well.

First, the variability model itself may evolve. Features may be added, removed, or modified as the capabilities of the system evolve. This in itself may not be a trivial task as the model must remain correct after modification. Potential errors have been identified in the past: false optional, redundant constraints, void model, etc. (Benavides et al., 2010). Specific tools and methods have been designed to identify such issues. Nonetheless, this highlights that making changes to large-scale variability models is not an easy

task. Previous work on the Linux kernel showed that addition and removal of features in its variability model is frequent (Lotufo et al., 2010).

Secondly, if features evolve, the mapping to their associated implementation artefact should evolve as well. This is less documented in the current state of the art. The work of Adams et al. showed how the build system of the Linux kernel, formalizing this association of features and files, evolves over time (Adams et al., 2008).

Finally, performing changes to the implementation of a variant-rich system is as challenging as performing changes to the implementation of any complex system. However, because of the fragmentation of concepts in various artefacts, and the fragmentation of artefacts by features, making changes to such artefacts can be difficult. As a result, artefacts of such systems must **co-evolve**. For instance, modifications to the presence conditions of code blocks expressed in macros should be done in accordance to the state of feature dependencies as expressed in the variability model of the system. As a result, engineers must manage changes affecting artefacts of different nature (variability model, implementation, build scripts), and guarantee that all those changes are consistent with one another. Inconsistencies between the variability in the variability model and its implementation will lead to a number of errors such as dead code blocks (Tartler et al., 2009), or variability specific bugs (Abal et al., 2014; Kenner et al., 2010; Medeiros et al., 2016).

3. CHALLENGES AND RESEARCH QUESTIONS

System maintenance and evolution is a key concern for complex and long-lived systems (Abran et al., 2014; Lehman, 1996). It is considered good practice to design a software system in such a way that future maintenance and evolution operations are as simple as they can be envisioned. A range of likely changes can be taken into account during the design phase such that the resulting design can accommodate them without requiring a complete redesign of the system. This ability of a system to accommodate for changes with a "reasonable" amount of transformation is known as the system *evolvability* (Breivold et al., 2008; Rowe and Leaney, 1997). In the past, actual software evolution scenarios, as performed by developers on complex systems in use, proved to be key elements. The seminal collection of "design patterns" proposed by the "Gang of Four" are the result of long working experience on complex long-lived systems. From their understanding of what happens in such systems, they were able to suggest design solutions that would be suitable for long term use. Unfortunately, in the context of highly variable systems, little information is known on how such systems evolve in practice.

As explained in Section 2, variant-rich systems are implemented using a set of heterogenous yet related artefacts. The evolution of such systems is then composed of changes to all those artefacts. From the work of Kenner et al. 2010 on code correctness, we know that a number of change scenarios are "safe" when affecting all those artefacts consistently. Considering single artefact changes as evolution scenarios for variant-rich systems is, therefore, insufficient. Gathering information on the evolution of such systems involves observing changes in a number of artefact, and regroup them comprehensively.

Since features are key design elements for variant-rich systems, we study the evolution of variant-rich systems through the evolution of their individual features. However, as mentioned earlier, in variant-rich systems, features are spread in a number of im-

plementation artefacts: the variability model, the mapping between features and assets, and finally the actual artefacts themselves characterizing the behaviour of the feature in question. This leads to the following overall research question:

Research question:
In the context of a variant-rich system, how do features evolve and how is that evolution reflected in the co-evolution of its concrete artefacts?

If we know the types of changes we are very likely to encounter in the future, then we are in a better position to evaluate design alternatives for the implementation of the concrete artefacts we need. If we know how a change is likely to cause changes across various artefacts of different nature, then we are in a better position to decide how to create and maintain the relationships between those artefacts. But the information currently available is not sufficient at this time.

Researchers already provide some information on those questions, such as broad statistics of variability model evolution, or through the analysis of code changes. However, to obtain clear information, we need to have information regarding the evolution of features over long periods of time, covering all types of development artefacts, with more details than currently available. Moreover, since we know that features are scattered among a number of different artefacts and we know that all those artefacts must be consistent with each other, we cannot restrict our understanding of feature evolution in variant-rich systems to their evolution in a single type of artefact. We need to consider how those artefacts change as a whole.

Performing such work will require a lot of information on how a variant-rich system evolves through the evolution of its features. Up until now, there are no automated means to aggregate changes pertaining to one feature in all of the artefacts over time. We need to find the means to obtain the necessary information with the higher level of details of changes in all relevant artefacts over a sufficient period of time to draw any conclusions. This motivated the core of the work presented in this thesis, and leads us to the following research goal:

Research goal:
To obtain a comprehensive view of feature evolution of variant-rich systems across heterogeneous artefacts and over a long period of time.

Through this work, we aim at making a significant step forward in the state of the art regarding the evolution and maintenance of variant-rich systems. More specifically, we aim at increasing:

- our ability to describe changes within each type of artefact
- our ability to describe complex scenarios affecting more than one type of artefact
- our knowledge on the methodologies and approaches that can be used to obtain this information

- the quantitative information available on occurrences of scenarios affecting more than one type of artefact

In order to reach this goal, we break down our work in several steps, based on the current state of the art on feature evolution in variant-rich systems.

VARIABILITY MODEL EVOLUTION

As shown by Lotufo et al. 2010 in the context of the Linux kernel, the number of features in the kernel is growing over time, and so does the number of feature-related constraints. This in turn makes the maintenance of such models difficult. Several methodologies were designed to assist engineers in such endeavours (Botterweck et al., 2010; Heider et al., 2012a). Similarly, several descriptions of changes occurring in variability models are available, although we note that, in most cases, the scenarios are the results of an analysis of *possible* changes, based on the observed models (Guo and Wang, 2010; Svahnberg, 2000). On the other hand, studies focusing on detailed changes observed in evolving product lines or other highly variable systems tend to focus on addition and removal of features (Neves et al., 2011; Passos et al., 2015), mostly based on observations of a small number of changes.

Our hypothesis is that existing studies, focusing only on addition and removal of features, miss relevant types of changes, namely modifications. By understanding how large variability models evolve, we obtain a better understanding of what operations are performed by developers on variability models and how those should be supported. This leads us to our first research question:

> **Research Question 1**: *What are the operations commonly performed on features in a large-scale variability model?*

VARIABILITY MODEL CHANGES AND IMPACT

In practice, variability models serve multiple purposes, but two of those are more relevant for the problem at hand. First, they are a formalization of each available feature and their presence condition. Secondly, they provide a compact representation of allowed configurations. This was highlighted in a number of studies (Benavides et al., 2005; Thum et al., 2011). When developers perform changes to the variability model, they may perform any number of changes to features or constraints, affecting both individual features and their respective execution context, as well as the set of available configurations (Thuem et al., 2009). Determining which capabilities are affected, and which product might see its behaviour affected by a change in a single feature can be hard to determine (Siy and Perry, 1998). Heider et al. 2012a propose to rebuild every configuration to identify the ones that may have been negatively affected. White et al. 2010 showed that, given a faulty feature model, one can identify the steps necessary to fix broken configurations.

However, checking changes to all possible configurations of a system, when the system contains a large number of features, can become intractable: in the general case, the number of possible configurations increases exponentially with respect to the number of available features. Moreover, if the system contains a large number of features, information regarding configurations may be difficult to interpret since each configuration may

contain many features (too many for humans to understand). Through ripple effects, changes to features may affect the capabilities of many sub-systems - or components of the system, and not all of them might be relevant for engineers.

We make the following assumptions. Changes to some parts of the system may require more care than others, for instance external interfaces of a system may be considered as more critical than internal components. Given a change to a feature, can we determine if "critical" elements of the systems are affected? Is a component, or subsystem be affected and if it is, what capabilities are changed?

We make the hypothesis that the variability model of a system is the right type of artefact to use to answer such questions. However, variability models can be very large which makes the understanding of changes at a configuration level hard to understand for humans. Moreover, the computation of all possible configurations to evaluate change impact is not feasible for large models. This raises the question of how to represent the variability of a large-scale variant-rich system in such a way that impact computation is possible, and thereby yielding change impact information on specific parts of the system. This leads to our second research question:

Research Question 2: *How can feature-level information be leveraged to assist engineers during change impact analysis in variant-rich systems?*

FEATURE-ORIENTED ARTEFACT CO-EVOLUTION IN PRACTICE

As mentioned in the previous section, variability implementation goes beyond features and their relationships. The scattering of features in the implementation, the variability model and the build mechanism is a necessity to support fine-tuned product derivation. As features evolve through changes in the variability model, other artefacts may be impacted as well.

The co-evolution of features and class diagrams was highlighted by Seild et al. 2012 during their study on software product line evolution. Hellebrand et al. (Hellebrand et al., 2014) studied the co-evolution of features in the variability model and feature-related code metrics (feature scattering and usage in pre-processor statements). Similarly, we can find a number of works on feature influence on build systems (Dietrich et al., 2012a; Nadi and Holt, 2012; Zhou et al., 2015). More recently, Passos et al. started an effort to describe complex feature-related co-evolution scenarios occurring on the Linux kernel (Passos et al., 2015). Through intesive manual analysis of a large corpus of commits (365), they isolated 23 scenarios. Those scenarios contain descriptions of changes occurring in the different artefacts: the variability model, the mapping between features and assets, and changes inside conditionally compiled code blocks. With this study, Passos et al. showed that some characteristics of feature implementation, such as the type of artefacts mapped to the feature, or the presence of conditionally compiled code blocks associated with a feature, were key elements to describe and therefore understand the scope of such change operations.

This last study consistitutes the state-of-the-art regarding feature oriented co-evolution of artefacts in variant-rich systems. However, being a manual study based on a relatively small number of commits, the results of this study cannot be generalized. Moreover,

while this study shows how co-evolution occurs in some cases, the manual nature of the analysis makes it difficult to replicate and further deepen our understanding of such changes. For instance, we lack information regarding the frequency of such changes and their prevalence in the evolution of systems over long periods of time. This leads us to our third research question:

> **Research Question 3**: *What role does co-evolution play in the evolution of features of variant-rich systems?*

From a technical perspective, complex co-evolution scenarios are challenging. Modifying each type of artefact requires a different expertise - if only to master the syntax and semantics used in that artefact, as well as their internal relationships. When hundreds of developers are involved in the development of a complex system, the question as to who performs complex evolution scenario arises.

If feature evolution may be performed by inter-related changes, it may be that the operations leading to a new version of a feature might be performed by more than one developer. Among all the developers working on such systems, we suggest that some may not have the expertise to perform such changes, and this should be reflected in the observed development practices. This leads us to our final research question:

> **Research Question 4**: *To what extent are developers facing co-evolution over the course of a release?*

4. RESEARCH METHOD AND EVALUATION

To answer our research questions, we studied the evolution of features in open source systems and in industrial context through case studies. Our first case study is the Linux kernel, a large-scale open source operating system, and the second is the Allura system, an x-ray machine developed by Philips Healthcare.

With more than 20 years of development, the Linux kernel contains today more than 13,000 features and is still actively developed. The kernel is present in one form or another in many types of systems: from watches to high-end computational clusters of servers. This makes the kernel a great example of a variant-rich system, with a long history of changes that we can learn from. In this period of time, we can observe complete life cycles of features, from their introduction until their retirement. Because of the popularity of the kernel, we can assume that the implementation techniques and maintenance practices, while not being perfect, are surely sufficient. We can consider this system as representative of a variant-rich, long lived and mature system, in which feature evolution is performed in a reasonable manner.

The second system on which we focus is the Allura system. It is an x-ray machine, with a large number of configuration options as well, although fewer than the Linux kernel. Developed by Philips Healthcare, the customization of the system is driven by customer needs, and in practice, each installed system is virtually unique. Because of the nature of the system (an x-ray machine), most elements are considered safety critical and

	R.Q. 1	R.Q. 2	R.Q. 3	R.Q. 4
Chapter 2	X	X		
Chapter 3		X		
Chapter 4	X		X	X

Table 1.1: Mapping between chapters and the research questions they address

must comply with strict regulations. Evolution of the system is closely monitored, both for internal impact as well as impact on the third party equipment that can be integrated with the Allura system. The Allura system is a good example of an industrial variant-rich system, where an understanding of feature changes and their impact is paramount for the evolution of the system.

The research presented in this work was conducted by applying elements of the design science research methodology (Hevner et al., 2004; Peffers et al., 2007). For each problem we tackle, for the purpose of either change extraction or impact computation, we built a prototype able to perform the main task at hand providing a concrete output with respect to the problem, either description of changes or description on their impact.

The prototypes themselves are quantitatively evaluated in terms of precision and recall when relevant. We present the methodology used to build the tools, in such a way that both the tools may be re-implemented and the overall experiment can be replicated. All prototypes developed during the course of this work are released as open source projects, for further use and repeatability purposes. For studies in which the tool generates data, the resulting datasets were also made available publicly. The nature of the data contained within the dataset is also described, along with the justification of its representation. The data collected with those prototypes are then used to derive information regarding feature evolution and feature changes in development artefacts.

The work on change impact performed in cooperation with Philips Healthcare was performed following the guidelines of an "industry-as-lab" experiment (Potts, 1993). The motivation behind the work was an actual industrial challenge that Philips engineers were facing. The execution of the research led to the development of a new design. In this case, we evaluated the suitability of the approach and its output by applying the approach to a number of industrial scenarios, and reviewed the results with domain experts.

In each chapter, we describe the scope and limitations of our approaches with respect to their generalization beyond the Linux kernel and the systems studied in the Philips case-study.

5. THESIS OUTLINE

This thesis is organized into chapters, each addressing at least one of the research questions presented in the previous section. An overview of the chapters and the questions they address is presented in Table 1.1.

In Chapter 2, we tackle the evolution of features at a variability model level. We present **FMDiff**, a model-based change extraction approach, with its associated implementation allowing us to identify between two tags in a version control system to extract fine-grained changes operated by developers on features within the variability model. We apply this approach to the Linux kernel and show how we managed to build a dataset comprised of feature changes occurring between the first and last commits of a release. Our evaluation shows that features in the Linux kernel evolved more through modification of existing features and their constraints than through addition and modification, shedding new light on evolution scenarios previously less explored. It also allowed us to show the suitability of model-based differencing for variability model change extraction.

In Chapter 3, we continue to work on variability model changes, focusing this time on change impact on valid configurations. We build a "multi-product line" representation of the variability model (a set of related variability models) of a complex hardware-software system, namely the Philips Allura X-ray system, and show how this representation can be used to assess change impact not on configurations of the system (of which there might be many, with many features), but on configurations of sub-systems - smaller, easier to understand. We also show how using such models we can observe the propagation of the change impact on the different configurations of each subsystem within the represented scope. In this case study, we modelled the variability of the system in terms of internal hardware, software elements, and external and publicly visible interfaces. Our evaluation shows how "multi-product line" representation of the variability of a complex system can be built in such a way that impact computation is feasible. We also show how this representation can be suitable for practical applications.

Chapter 4 is a presentation of the **FEVER** approach. This model-based change extraction approach is an extension of what was made possible by **FMDiff** but now includes model-based change extraction from build system and source code. This approach allowed us to build a large dataset of feature change information and enabled us to provide the first set of quantitative information regarding co-evolution of artefacts in the context of feature evolution. The evaluation shows that a model-based approach can be used to extract feature-related information from different types of artefacts. The evaluation of the collected data shows that while co-evolution occurs systematically over time, complex change scenarios are not the most frequent. We also show that a minority of developers face complex co-evolution scenarios during development activities.

Finally, in Chapter 5, we reflect on the collected results and our research questions. We then discuss their potential implication on software engineering practices and research.

6. ORIGIN OF CHAPTERS

The chapters of this thesis are all based on peer-reviewed publications, accepted in software engineering conferences and journals. Each chapter is self contained, comes with its own set of contributions. While this helps to read chapters independently for each other, this implies some overlap between the different chapters, especially in their introduction and motivations.

The author of this thesis is the main contributor of the work presented in each chapter. In addition, Chapter 2 includes a postscript section detailing the author's current

perspective on the work presented therein.

- **Chapter 2** is mostly based on work published in the Springer journal "Software and Systems modelling" (SoSyM) in May 2015 authored by Dintzner, Van Deursen and Pinzger. An earlier version of this work was published as part of the Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS workshop) in 2013. The postscript of this chapter is based on the work of Rothberg et al. published as part of the Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS workshop) in 2016.
- **Chapter 3** This chapter first appeared in Proceedings of the 14th Conference on Software Reuse (ICSR'15) authored by Dintzner, Kulesza, Van Deursen and Pinzger.
- **Chapter 4** is based on work submitted to the journal of "Empirical Software Engineering" (submitted in October 2016, currently under review) authored by Dintzner, Van Deursen and Pinzger. An earlier version of this work appeared in the proceedings of the 13th International Conference on Mining Software Repositories (MSR'16), authored by Dintzner, Van Deursen and Pinzger.

2

Analyzing the Linux Kernel Feature Model Changes Using FMDiff

To improve is to change; to be perfect is to change often.

Winston Churchill

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

Using this information, we performed an exploratory study of changes occurring in the Linux kernel feature model. We show that modifications of existing attributes and constraints of features play a major role in the evolution of the Linux kernel feature model, and yet such changes are often overlooked by current research. Finally, by comparing the evolution of the different architecture specific feature models, we show that 10 to 50 % of feature changes performed in a given release affect the capabilities of all of them, thus making generalization of observations on feature evolution from one architecture specific feature model to others difficult.

This chapter was originally published as “Analyzing the Linux Kernel Feature Model Changes Using FMDiff”, in the Springer journal "Software and Systems modelling" (SoSyM) in May 2015 authored by Dintzner,Van Deursen and Pinzger.

1. INTRODUCTION

Software product lines are designed to maximize re-use of development artefacts while diminishing development costs, through the identification and formalization of what is common and variable between different members of a product family (Clements and Northorp, 2002). 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 in a software product line influences its architecture, implementation techniques, and applicable methods to instantiate products from a set of assets (source code, scripts, resources).

Over time, as a software product line evolves, features are added, removed, or modified and the associated assets are updated accordingly. However, software product lines are often large-scale, long lived systems and the number of features in a product line increases over time. The sheer size and the number of variable components in such systems make evolution operations difficult and error-prone.

Because of the pervasive effect of features on the design and architecture of product lines, the evolution of a software product line is linked to the evolution of its feature model. Recent research on co-evolution of feature models and other software assets in product lines suggests that maintenance operations can be facilitated by using feature changes as a starting point (Neves et al., 2011; Passos et al., 2013). Feature model evolution and feature transformations have been extensively studied in the past (Guo et al., 2012; Paskevicius et al., 2012; Svahnberg, 2000; Thuem et al., 2009). These studies provide insight on which operations may occur on features, detailed examples of specific transformations occurring on large scale product lines, and the evolution of feature model structural metrics (number of leaves, nodes, constraints).

Unfortunately, from this body of knowledge, gathering a detailed view of all changes occurring in practice on a large scale feature model is difficult. Previous work showed that knowledge about fine-grained changes of software artefacts could facilitate software maintenance. Using fine-grained changes of source code, good results were obtained in the area of bug prediction (Giger et al., 2012). In the context of software product lines, fine-grained information about feature changes, down to feature attribute values, could be a major asset to enable safe evolution of such systems.

With more than 10,000 features and over two decades of development history, the Linux kernel is a popular choice of system for the study of the evolution of large scale, industrial-grade product lines. Several studies (e.g., (Israeli and Feitelson, 2010; Lotufo et al., 2010)) 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 structure or number of leaf features in each release, as means to illustrate the evolution of both the kernel and its feature model.

Yet, the details of changes occurring on the Linux kernel features are not known. Existing feature model evolution studies focus essentially on the addition and removal of features (Lotufo et al., 2010; Neves et al., 2011; Passos et al., 2013), but there is no evidence that such changes are the most frequent on industrial feature models. By gathering fine-grained information about Linux feature changes, we can verify whether addition and removal of features are the change operation that weighs the most in the evolu-

tion of the Linux kernel feature model and answer the following research question:

RQ1: What are the most common high-level operations performed on features in the Linux kernel feature model?

Moreover, studies of the Linux kernel mostly focus on a single hardware architecture (often X86) and, on occasion, extrapolate their findings to other architectures (Lotufo et al., 2010). By obtaining details of feature changes, we will be in a position to assess whether such extrapolation holds when studying the evolution of the Linux feature model, and answer this second research question:

RQ2: To what extent does a feature change affect all architecture-specific feature models of the Linux kernel?

This paper is a revised and extended version of our work on the extraction and classification of feature model changes in the Linux kernel (Dintzner et al., 2013). We present first our classification of feature changes occurring in the Linux kernel feature model based on the Kconfig language[1] and an improved version of the corresponding tool, `FMDiff`, to extract them. Our classification describes feature changes on three different levels of granularity, feature model level operation (adding, removing, and modifying features), feature statement changes, down to feature attribute value changes. `FMDiff` uses `Undertaker` (Tartler et al., 2011, 2009) to extract the Linux feature model and the EMF Compare[2] diff algorithm to compare two subsequent versions. We evaluate our approach by comparing changes captured by our tool and changes applied to Kconfig files and vice versa. The results show that `FMDiff` captures a large majority of changes applied to Kconfig files and provides a more comprehensive view of feature changes than what could be obtained by looking at Kconfig file textual differences.

We further extend our work by using the classification and `FMDiff` to build a larger dataset of feature changes occurring on the Linux kernel feature model and use it to answer our two research questions. We build a dataset comprised of feature changes obtained by extracting the change history of more than twenty architecture-specific feature models over fourteen releases of the Linux kernel, from release v2.6.39 until release v3.14. Using the collected data, we show that modifications of existing features is a predominant operation on the kernel feature model. Finally, we show that, despite a large proportion of common features between all architecture-specific feature models, a large majority of feature changes only affect a subset of them.

The main contributions of this paper are: 1) a feature model change classification scheme, focused on the Linux kernel Kconfig language; 2) a tool-supported approach to extract and classify automatically feature model changes from a versioning system, evaluated using ten releases of the Linux kernel; 3) evidence that changes to existing features constitute a large proportion of transformations of the Linux feature model; 4) evidence that significant differences between the evolution of the different architecture-specific feature models of the Linux kernel exist. Both our tool and our dataset are available for download.[3]

The remainder of this paper is organized as follows. Section 2 provides some background information on the Linux kernel feature model and its reconstruction. We present

[1] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
[2] http://www.eclipse.org/emf/compare/
[3] https://github.com/NZR/Software-Product-Line-Research

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 FMDiff and the use of fine-grained feature changes to analyze the evolution of software product lines in Section 8 Section 9 presents related work. Finally, we conclude this paper and elaborate on potential future applications of FMDiff in Section 10.

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 using the kdump tool. In this section, we present general information regarding the Linux kernel FM, the basic Kconfig concepts and the output format generated by kdump.

2.1. THE LINUX KERNEL FEATURE MODEL

Linux users can tailor their own kernel with Xconfig (among other tools), the kernel configurator. This tool displays available configuration options in the form of a tree, and as the user selects or unselects options, the tree is updated to show only options that are compatible with the current selection. In this context, configuration options can be assimilated to features and the set of options with their constraints to a feature model (Sincero et al., 2007; Sincero and Schröder-Preikschat, 2008).

Such tools use the textual descriptions of the Linux features contained with Kconfig files as an input (detailed in Section 2.2), 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 (Lotufo et al., 2010; Nadi and Holt, 2012). In this study, we use this specific approach when rebuilding the Linux FMs and analyzing FM changes.

2.2. KCONFIG

Features and their composition rules, which denote cross-tree constraints in FMs, are specified using the Kconfig language (see also (Sincero et al., 2007) and (Sincero and

```
1 if ACPI

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

Listing 2.1: Example of a feature declaration in Kconfig

Schröder-Preikschat, 2008)). Listing 2.1 exemplifies a feature declaration in the `Kconfig` language.

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

Kconfig supports two types of dependencies. The first one represents pre-requisites, using the `depends` (or `depends on`) statement followed by an expression of features (see line 6). If the expression is satisfied, the feature becomes selectable. The second one, expressing reverse-dependencies, are declared by the `select` statement. If the feature is selected then the target of the `select` will be selected automatically as well (`POWER_SUPPLY` is the target of the `select` statement on line 7). The `select` statement may be conditional. In such cases, an `if` statement is appended to the `select` statement. `depends`, `select` and constrained `default` statements are used to specify the cross-tree constraints of the Linux kernel FM (She et al., 2010). A feature can have any number of such statements.

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

Finally, `Kconfig` offers the possibility to define 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. Like "if" statements, constrains defined on menus and menuconfigs are applicable to all elements within.

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

Listing 2.2: Representation of the feature declaration of Listing 2.1 in .rsf format

2.3. FEATURE MODEL REPRESENTATION

A prerequisite to our approach is being able to extract feature definitions from Kconfig files. For this, we use an existing Linux tool, `kdump`, to translate Kconfig features into an easier to process format. This tool has been used in other studies of the Linux variability model, where `kdump` output is used by Undertaker(Tartler et al., 2009) to determine feature presence conditions. `kdump` produces a set of ".rsf" files[4], each one containing an architecture-specific FM, i.e., an instance of the Linux FM where the choice of hardware architecture is predetermined. Listing 2.2 shows the example of the feature declared in Listing 2.1 in rsf triplets as output by `kdump`.

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.

`kdump` eases feature extraction but modifies their declaration. Among the applied modifications, two are most important for our approach: first, `kdump` flattens the feature hierarchy and then, it resolves features `depends` statements. Concerning the flattening of the hierarchy, `kdump` modifies the `depends` statement of each feature to mirror the effects of its hierarchy. For instance, `kdump` propagates surrounding `if` conditions to the `depends` statements of all features contained in the if-block. This explains the addition of `ACPI` to the condition of the `depends` statement on line 5 of Listing 2.2. Concerning the resolution of `depends` statements, `kdump` propagates conditions expressed in the `depends` statement of a feature to its `default` and `select` conditions. This explains the condition `X86 && ACPI` that has been added to the select (`ItemSelects`) and default value (`Default`) statements.

3. CHANGE CLASSIFICATION

We aim at classifying feature changes occurring in the Linux kernel feature model (FM). Existing feature change classifications do not consider some specificities of the Kconfig grammar (e.g., select relationships with conditions, default value, or visibility). To capture as accurately as possible changes in such statements, we introduce a new classification.

[4] Rigi Standard Format is a simple notation for annotated triplets originating from the Rigi reverse engineering tool. http://www.rigi.cs.uvic.ca/downloads/rigi/doc/node52.html

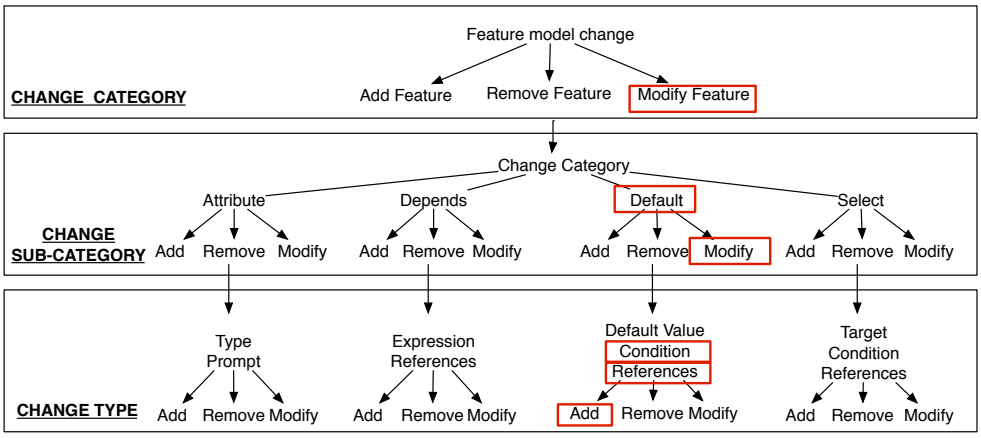

Figure 2.1: FMDiff scheme to classify changes in feature models on different levels.

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 Kconfig statement, such as a `default` statement, and the change operation applied i.e., addition (`ADD`), removal (`REM`), or modification (`MOD`). This classification is based on the Kconfig grammar[5], to identify which feature attribute may change and the relationship between them. We then consider that each attribute and feature may be added, removed or modified. Figure 2.1 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

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

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.

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

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

Regarding our classification scheme, 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

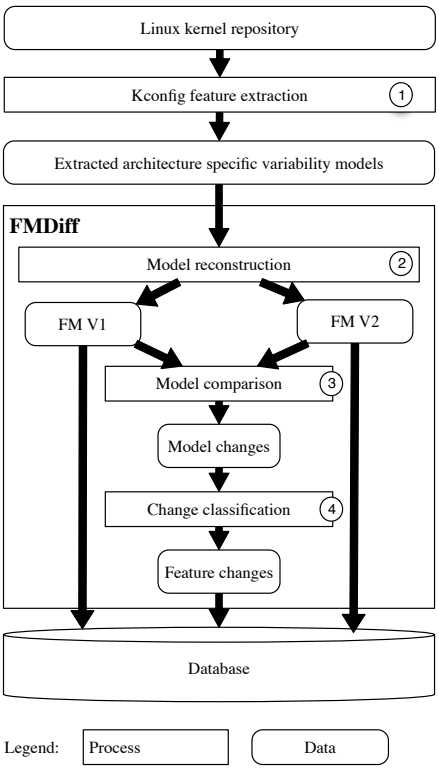

Figure 2.2: Change extraction process overview

change type `ADD_TYPE` can only occur in the context of a feature creation, i.e., with the *change category* `ADD_FEATURE`.

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

4. FMDIFF

In this section, we present our approach to automate feature change extraction and the tool that supports it: `FMDiff`. We then evaluate our approach by comparing changes captured using `FMDiff` with changes performed on Linux Kconfig files.

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 2.2.

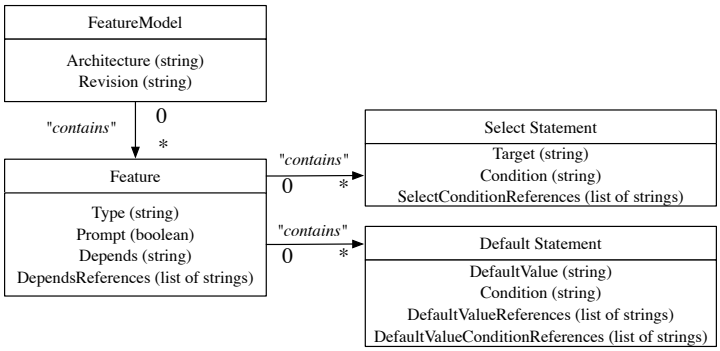

Figure 2.3: FMDiff feature metamodel

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.[6] Next, we use the `Undertaker` tool, that provides a wrapping of `kdump`, to extract architecture-specific FMs for each version. Undertaker outputs one ".rsf" file per architecture per version, in the format described in Section 2.

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

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

FMDIFF FEATURE MODEL CONSTRUCTION

As a second step, we reconstruct FMs from two subsequent versions of a ".rsf" file. `FMDiff` compares FMs that are instances of the meta-model presented in Figure 2.3.

`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, tri-state, integer, etc.), and prompt attribute. In addition, each feature contains a `Depends` attribute representing the `depends` statements of a Kconfig feature declaration. All features referenced by the `depends` statement are stored in a collection of feature names, called `DependsReferences`.

Each feature can have any number of `Default Statements`, containing a default value and its associated condition. Furthermore, a feature can have any number of `Select`

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

Statements containing a select target and a condition. The condition of both statements is recorded as string by the attribute Condition. The features referenced by the condition of each statement are stored in the collection DefaultValueReferences or SelectReferences respectively.

The .rsf output also allows a feature to have multiple depends statements but, in our meta-model, we allow features to have only one. In the case where FMDiff finds more than one for a single feature, it concatenates those statements using a logical AND operator. This preserves the Kconfig meaning of multiple depends statements.

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

We store the reconstructed FMs and features in our database for later use. This same representation is also used in the next step: the comparison of FMs.

COMPARING MODELS

For the comparison of two FMs, FMDiff builds upon the the EMF Compare[7] framework. EMF Compare is part of the Eclipse Modeling Framework (EMF) and provides a customizable "diff" engine to compare models. It is used to compare models in various domains, like interface history extraction (Romano and Pinzger, 2012), or IT services modelling (Hölzl and Feilkas, 2010), and is flexible and efficient. EMF Compare takes as input a meta-model, in our case the meta model presented in Figure 2.3, and two instances of that meta-model each representing one version of an architecture-specific Linux FM. EMF Compare outputs the list of differences between them.

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

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

EMF Compare generates a list of the differences between the two models, expressed using concepts from the FMDiff feature meta-model. For instance, a difference can be

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

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.

CLASSIFYING CHANGES

The last step of our process consists in translating the differences obtained by EMF Compare into feature changes as defined by our classification scheme.

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

The results are stored in a relational database. We record for each feature change: the architecture and 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 (completeness) 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 complete if it contains all changes seen in Kconfig files, and its data interesting if it provides the information needed to understand the changes in the Linux feature model (FM). We start by describing the dataset used for the evaluation, and then assess them separately before discussing the obtained results.

DATA SET

Using Git, we retrieve the history of the Linux FM. Lotufo et al. highlight that at random points in time, the Linux FM is not necessarily consistent(Lotufo et al., 2010). To minimize such issues, we extract feature changes between official Linux releases. For all releases of the Linux Kernel from 2.6.28 to 3.14, 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).

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.

COMPLETENESS

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

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

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

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

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

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

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

INTERESTINGNESS

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

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

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

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

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 kdump's output. We obtained 26 *matches*, 79 *indirect matches* and finally 2 features were renamed and those changes were successfully captured as deletion and creation of a new feature. Among the *indirect matches*, 61 are due to hierarchy expansion and 18 due to depends statement expansion on other attributes.

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

79 feature changes captured by FMDiff could not be directly linked to feature changes in Kconfig files but to changes in the feature hierarchy or other feature attributes. We argue that even if FMDiff data does not always reflect the actual modifications performed by developers in Kconfig files, it captures the effect of the changes on the Linux FM. In fact, FMDiff data provides better 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.

4.3. DISCUSSION: FMDIFF

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

CAPTURING CHANGES

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

Developers and maintainers modifying Kconfig files can use our tool to assess the

effect of the changes they perform on 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.

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

THREATS TO VALIDITY

Internal validity The evaluation of the tool was done by manually inspecting changes in Kconfig files and recorded changes in `FMDiff`. Like most manual processes, it is error prone. We recorded comparison and matches and we share the sample and results on our website for further validation.[8]

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

For our evaluation, we compared the changes in Kconfig files with their respective records in FMDiff database. In total, for this experiment, we checked 229 database entries (121 during the first step and 108 during the second). The complete database used for this experiement contains 2,734,353 records. For a confidence level of 95%, we can consider our results with a confidence interval of 6.48. Out of the 229 checked entries, 223 were valid. We argue that, while our sample is relatively small, and the confidence interval quite large, our conclusions regarding change types and architecture-specific feature evolution can reasonably be relied upon.

External validity Our change classification and tool are tightly linked to the Kconfig language. While a mapping between Kconfig and more generic FMs (such as FODA) exists (She et al., 2010), we did not investigate its usage to generalize our approach. This work is currently limited to product lines using the Kconfig language as a means to describe their features.

5. USING FMDIFF TO UNDERSTAND FEATURE CHANGES IN THE LINUX KERNEL FEATURE MODEL

`FMDiff` captures changes occuring 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

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

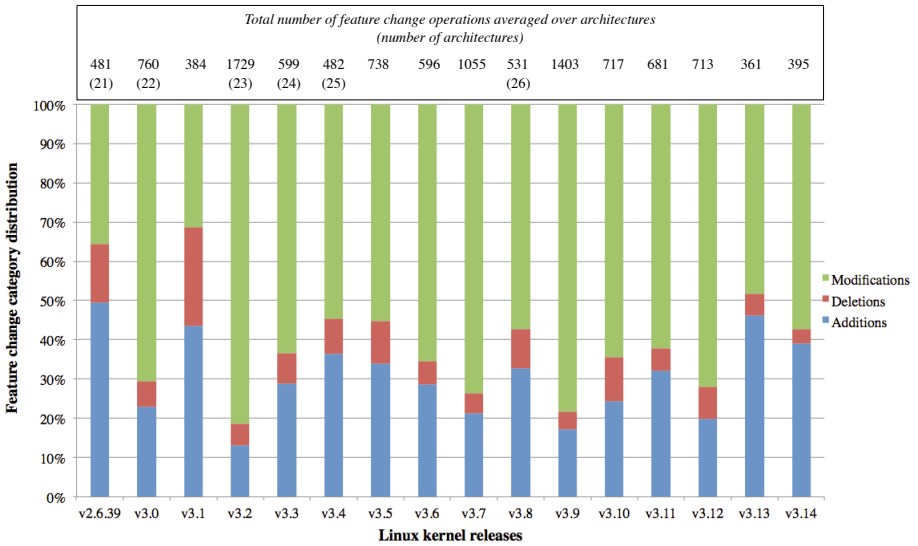

Figure 2.4: Evolution of the feature change category distribution (averaged over architectures)

and study the pervasiness of feature changes across the multiple architecture-specific FMs of the kernel, and to answer the research questions as raised in the introduction.

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 (Neves et al., 2011; Passos et al., 2013; Seidl et al., 2012), 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 high-level 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 2.3 shows an example of such query, giving us the number of features modified during release v3.0 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 2.4, during release 3.0, the average number of feature changes in architecture-specific FMs were 760. About 70% of those changes are modifications of ex-

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

Listing 2.3: Example of query on FMDiff data: modified features in release 3.0

isting 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 2.4, 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 occuring 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). To obtain a better understanding of the evolution of a SPL with respect to its FM, it is necessary to understand how simple feature transformations, such as attribute or constraints changes are implemented. In summary, to answer RQ1, the most frequent high-level change operations performed on the Linux feature model are modifications to existing 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. Further studies on different systems are necessary to generalize this finding to other SPLs, in other domains.

5.2. EVOLUTION OF ARCHITECTURE-SPECIFIC FMS

The Linux kernel feature model (FM) has been extensively studied as an example of large scale software product line. 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 (Lotufo et al., 2010). 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 a up to 60% of their features

(Dietrich et al., 2012b), and that the growth rate of architecture-specific FMs are similar (Lotufo et al., 2010).

In this section, we compare the evolution of the different architecture-specific FMs. Our aim is to assess how similar their evolution is and if, at a fine-grained level, the assumption presented above still holds; and answer our second research question: *RQ2: To what extent does a feature change affect all architecture-specific FMs of the kernel?*

MOTIVATION

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 v3.0, feature ACPI_POWER_METER was removed and replaced by SENSORS_ACPI_POWER contained in another code module.[9] 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 - and perhaps others. Given the commit message, it is unclear whehter 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 v3.0, feature X86_E_POWERSAVER pre-existing in the X86 architecture was added to other architectures and its attribute modified. By searching in Git history, we identified the commit[10] 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 v3.0, the depend statement and select condition attributes of this 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 thourough testing practice would require validating a change for all architectures. When a developer modifies the behavior 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 to the FM affect an architecture-specific feature, or if an architecture-specific change is applied to a feature. However, the cross-compilation process is non-trivial.[11] 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,

[9] commit: 7d0333653840b0c692f55f1aaaa71d626fb00870
[10] commit: bb0a56ecc4ba2a3db1b6ea6949c309886e3447d3
[11] Linux cross-compilation manual:
http://landley.net/writing/docs/cross-compiling.html

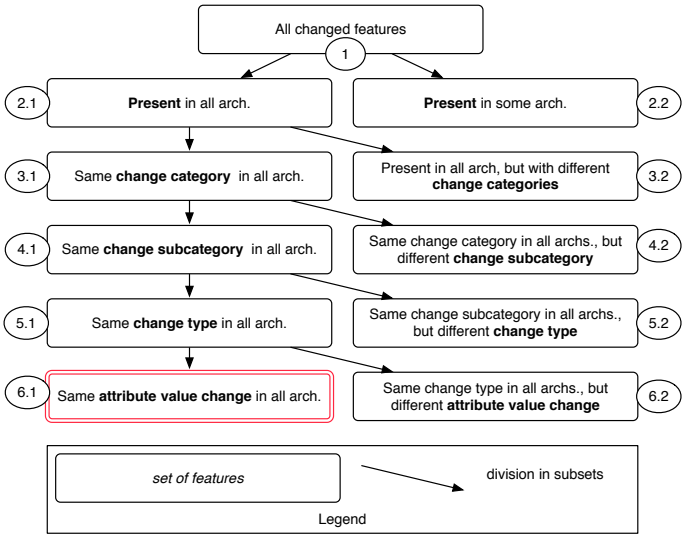

Figure 2.5: Extracting feature changes affecting all architectures

> "Untested as I don't have a cross-compiler." [12]
>
> "We have only tested these patchset on x86 platforms, and have done basic compliation tests using cross-compilers from ftp.kernel.org. That means some code may not pass compilation on some architectures." [13]

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." [14]

METHODOLOGY

To analyse the discrepancy between the evolution of the different architecture-specific FMs, we compare the changes occuring on the features of the different FMs during the same release. We proceed as shown in Figure 2.5.

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 2.5). 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 2.5.

[12] commit: 2ee91e54bd5367bf4123719a4f7203857b28e046
[13] commit: cfa11e08ed39eb28a9eff9a907b20913020c69b5
[14] https://lkml.org/lkml/2011/7/26/490

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 features that are seen as changed exactly in the same way in all archictures (set "6.1" in Figure 2.5). 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.

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 v3.3, 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.

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.[15]

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

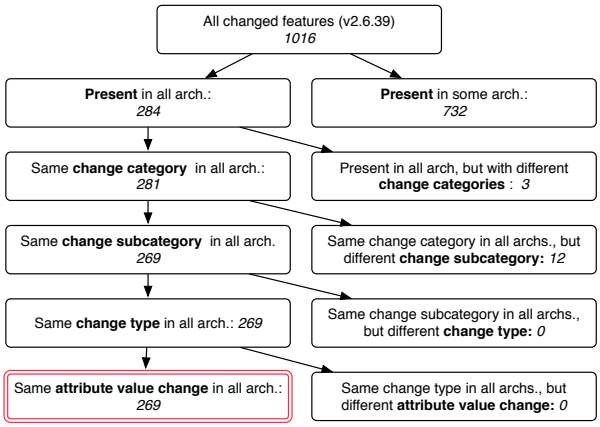

Figure 2.6: Example of architecture evolution comparison for release v2.6.39

RESULTS

By applying the methodology described in this section for a single release, we obtain the information depicted in Figure 2.6. We can read this figure as follows: in release v2.6.39, 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 FM 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 2.1. In this table, each release column is read like diagram Figure 2.6, presenting the number of changed features affecting all (generic) or some (arch-specific) architecture-specific FMs, decomposed by change operation granularity - from touched to attribute value changes. From this table, we learn the following.

First, the total number of changed features in each release, shown in the second row of Table 2.1, is very variable. Over the studied period of time, the release with the smallest amount of changed features is v3.1, with only 567 changed features, and the release with the largest number of changed feature is release v3.11, 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 2.1 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

Release	**v2.6.39**		**v3.0**		**v3.1**		**v3.2**	
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	**v3.3**		**v3.4**		**v3.5**		**v3.6**	
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	**v3.7**		**v3.8**		**v3.9**		**v3.10**	
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	**v3.11**		**v3.12**		**v3.13**		**v3.14**	
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 2.1: Quantitative comparison of generic and “architecture-specific” feature changes

is touched in all architectures if it is seen as added,removed or modified in all architectures - regardless of the extact change type (as shown in the third row of Table 2.1.

Finally, no features have architecture specific change type and attribute value changes. In all release, 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, we do not isolate changes made to all attributes. This causes small discrepancies in the values shown in Table 2.1. For instance in release v3.4, 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 v3.11 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[16] modifying the CRIS architecture configuration file (/arch/cris/Kconfig). The modification, shown in Listing 2.4, removed the inclusion of a specific set of drivers and replaced it by the inclusion of all standard drivers. Such refactoring can be observed in release v3.11 for the CRIS, ARM and H8300 architectures.

In each cases, relying on the common drivers, without cherry picking, involves relying on the structure of that Kconfig file including its menu structures. As a result the exact presence condition of included features are changes to reflect this: you now need to select the over-arching "Device Drivers"" feature before selecting any of the subfeatures. So, while the intent to make more drivers available for a given architecture and simplify the Kconfig file hierarchy, the result is an actual refactoring of the variability model involve large scale modification of features presence condition.

From a architecture-specific perspective, such refactoring lead to a large number of modified features (for the drivers that were already included, but with a different hierarchy), as well as a large number of added features (for all the drivers that were not previously cherry picked for that architecture).

Finally, we consolidate our results in Table 2.2. 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-specific FMs. We can read Table 2.2 as follows: in release v3.12, 47.93% of the 1406 changed features were seen as changed consistently in all architecture-specific FMs of the Linux kernel.

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?*

[16] commit: acf836301e4b8f3101c5f83e4a52dbb6c3899314

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

 source "fs/Kconfig"

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

Listing 2.4: Extract of the diff of file "/arch/cris/Kconfig" in release v3.11

Linux Kernel release	Total number of changed features	% of changed features affecting all architectures
v2.6.39	1016	26.47
v3.0	1020	58.43
v3.1	567	35.62
v3.2	2361	39.00
v3.3	946	24.10
v3.4	778	32.39
v3.5	1103	39.16
v3.6	823	34.14
v3.7	1285	29.09
v3.8	963	29.38
v3.9	1773	57.75
v3.10	1299	32.10
v3.11	4556	8.12
v3.12	1406	47.93
v3.13	620	52.58
v3.14	704	53.12

Table 2.2: Evolution of the ratio of feature changes impacting consistently all architecture supported by the Linux kernel.

The data shown in Table 2.2 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 conclusion on the evolution of the overall Linux FM. Table 2.1 emphasizes that most feature changes might not even be seen in other architectures. It is interesting to note that, during release v3.11, while 4556 features were changed during the release but the average number of changed features per architecture is 681 (see Figure 2.4). This further supports our assumption that architecture-specific FMs evolve differently.

However, Table 2.1 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 v3.12, where among the 678 changed features that affected all architectures, all had the same *change category*, *change subcategory*, *change type* and at-

tribute 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 v3.11 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.

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: ON THE USE OF FINE-GRAINED FEATURE CHANGES

The main objective of this paper is facilitating the maintenance and evolution of large scale software product lines (SPLs). As shown in previous studies, changes to feature models (FMs) are coupled with changes in their implementation. To some extent, given a change to a FM, one could determine in a systematic way how to update the associated implementation (Alves et al., 2006; Passos et al., 2013). In this section, we reflect on the usage of fine-grained feature changes and our tool in the context of SPL maintenance.

6.1. USING FMDIFF TO STUDY FM EVOLUTION

As mentioned in Section 4.3, `FMDiff` captures accurately a large majority of feature changes applied on 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). This provides insight in the transformations applied to each individual feature.

Changes made to FMs affect the set of allowed configurations (Thuem et al., 2009), e.g.,, allowing the choice of a driver for a given architecture. While we capture the change performed on each individual feature, identifying the impact of a feature change on possible configurations of a SPL is outside the scope of this study. However, such information could be important when relating feature changes to implementation changes, because the implementation of all features involved in new configurations might have to be adapted to accommodate new runtime or buildtime constraints and their definition might remain the same. Given the size of the Linux kernel FM (more than 12,000 features), computing all possible configurations before and after a change to identify changes in allowed configurations is not practical. A potential solution would be to apply model checking approaches to perform semantic differencing between two versions of a FM (Gheyi et al., 2006), in order to identify samples of configurations allowed in one version but not in the other. Furthermore, fine-grained feature changes and configurations changes could be considered as complementary descriptions of the same change operations. We will explore the possibilities offered by such approaches in future work.

The dataset we built for this study contains changes occurring between official releases of the Linux kernel. Currently, `FMDiff` uses a list of Git tags to identify which version of the Linux FM should be extracted for comparison. This dataset gives us a comprehensive view of changes in each release, but in other contexts it might be interesting to extract changes occurring over shorter periods of time. The implementation of `FMDiff`

is flexible enough to allow this. By simply replacing the Git tags by Git commit identifiers, we can extract feature changes occurring over a shorter time period (down to 1 commit) with no modification to the differencing heuristics.

6.2. TOWARDS IMPLEMENTATION CHANGES

In the Linux kernel, changes to feature attributes and cross-tree constraints are the most frequent. To extend existing work on co-evolution of FM and implementation, we need to relate such common changes with implementation changes. While it might be possible to determine where the attribute value of a feature is used in the implementation of a SPL and assess the effect of a value change, inferring the impact on the implementation of changes in cross-tree constraints might be more difficult.

In Linux, constraints are expressed using two different statements 'depends' and 'select' and should represent two different types of dependencies. But in practice, there is little difference between the usage of those statements.

> "We _are_ sprinkling tons of selects all around the Kconfigs. And we're doing it inconsistently - nobody seems to agree on when to use 'select' and when to use proper dependencies."[17]

This means that we cannot rely on the difference between 'depends' and 'select' to relate a feature change to potential implementation changes. Moreover, features in Linux represent a wide range of concepts: from code maturity information (the EXPERIMENTAL feature for instance), to code library (e.g.,, GPIO). As a consequence, a new cross-tree constraint between two features might be implemented differently depending on which features are involved in the constraint.

To relate code and feature changes and use that information to facilitate maintenance, we might need to define more precisely the nature of the relationships between features, or the nature of the features themselves. While we do not provide answers to this problem, we believe that, in the context of the Linux kernel, additional information can be mined from the implementation itself prior to change extraction. For instance, the folder structure of the Linux kernel allows us to identify which features represent device drivers. We will explore such possibilities as part of our future work.

6.3. ARCHITECTURE-SPECIFIC EVOLUTION

The comparison of architecture-specific FMs evolution showed us that there are feature changes that affect the various architecture-specific FMs in different ways. In the case of the Linux kernel, rebuilding a single architecture-agnostic FM is difficult. This implies that future work on the evolution of the Linux kernel FM should mention which architectures have been studied - as the same observation might not be true in all of them.

Knowing that a feature change affected only certain architecture-specific FMs might provide information about the potential changes in the SPL implementation. Intuitively, the implementation of a feature change affecting only certain architectures should only affect the implementations of those architectures. To the best of our knowledge, this has

[17] http://www.gossamer-threads.com/lists/linux/kernel/729689#729627

not been demonstrated yet, but if so then fine-grained feature changes in architecture-specific FMs could be useful to verify the consistency between implementation and variability model changes in SPLs.

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 subproduct-lines. For instance, in the automotive domain, one can use this approach to identify which feature changes affected the variability model of the "sport" and "family" variants of a car, where each variant is a product line on its own. However, the results found during this study are only applicable to the Linux kernel FM and should not be generalized without further investigations.

7. RELATED WORK

The Linux kernel has been used as an example of an evolving software product line many times in the past. Israeli et al. 2010 show that the Linux kernel evolution follows some of Lehman's Law of software evolution (Lehman, 1996), namely the continuing growth by measuring the number of lines of code over time. Lotufo et al. 2010 study the evolution of the Linux kernel variability model over time through FM structural metric evolution (model size, number of leaves, etc.). They show in their study that the number of features and constraints 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 feature model (FM) structure, properties, and evolution, several research teams have developed tools to reconstruct a FM from Kconfig files. LVAT (She et al., 2011) and `Undertaker` (Dietrich et al., 2012a; Sincero et al., 2010b; Tartler et al., 2011) are the main examples of such tools. We chose to rely on `Undertaker` for its convenient wrapping of `kdump`, allowing us to use the same tools that are also used by the Linux kernel development team. LVAT could have allowed us to capture the feature hierarchy. However, `kdump` flattening of the hierarchy facilitated the capture of feature hierarchy changes through changes of `depends` statements.

Several FM change classifications have been proposed in the past. In his thesis, Paskevicius describes (Paskevicius et al., 2012) several transformations that can be applied on a FM. Similarly, FM change patterns have been identified by Alves et al. in 2006 and Neves et al. in 2011. In their study of the co-evolution of models and feature mapping, Seidl et al. in 2012 also describes a set of operations applied to FMs. Thüm et al., in 2009, 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. in 2013; 2012 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. in 2011 a `depends` statement can either be interpreted as a cross-tree constraint or a hierarchy relationship, so we cannot automatically map changes of `depends` statements in other change classifications. Second, `FMDiff` is able to capture changes in feature attributes which are not considered by these classifications.

In recent work, Passos et al. built a dataset of feature changes of Linux (Passos and Czarnecki, 2014). 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

In this paper, we presented a classification scheme to categorize changes in the Linux feature model and the `FMDiff` tool to automatically extract these changes from two versions of Kconfig files declaring the Linux feature model. We evaluated our approach by manually validating the changes extracted by FMDiff from ten releases of the Linux kernel. The results show that our approach can capture feature changes accurately. A comparison between the information on changes obtained with `FMDiff` and the information obtained through manual analysis of the textual differences between Kconfig files highlighted that our approach provides a more comprehensive view on feature changes. Using the data captured by `FMDiff`, we observed that modifications to existing features (attributes and constraints) account for a large proportion of operations performed on the Linux features. Finally, we studied how feature changes affected architecture-specific feature models of the kernel and showed that, despite the large number of features shared between them, few feature changes affect all of them.

As a next step, we plan to detail our studies on the evolution of the Linux feature model by analyzing the fine-grained change types. Using the data acquired by `FMDiff`, we will answer questions such as what are the most frequent types of changes performed in the Linux feature model and which features and parts of the feature model are changing frequently. While we have shown here that feature changes do not affect equally all architecture-specific feature models of the Linux kernel, they might affect equally some of them. We believe that our dataset contains the necessary information to identify groups of architecture-specific feature models that evolve more consistently than others. Another direction of our future research is to investigate the impact of feature changes on other variability spaces, such as build and source code variability. For instance, we plan to explore how feature changes ripple through the Linux kernel implementation.

9. POST-SCRIPT: ON FEATURE MODEL CHANGE EXTRACTION

This work on variability model change extraction presented in this chapter motivated further research on the evolution of the Linux kernel features. Rothberg et al. continued the exploration of methodologies and approaches to understand how features evolve in such large scale systems (Rothberg et al., 2016). Our approach uses a semi-structured syntactic approach to finding differences, whereas Rothberg et al. use a semantic approach. In this section, we reflect on the work of Rothberg et al. and its implication on approach and findings.

9.1. SYNTACTIC AND SEMANTIC CHANGES OF VARIABILITY MODELS

We provide first a short introduction to the concepts of syntactic differencing described in this chapter and semantic differencing approaches. A variability model represents a compact representation of a set of configurations. It is, however, formalized as a set of formatted string within a file. When a change occurs, a developer modifies features in that model, and the change is performed by modifying character strings within that file. If we want to provide a description of the change, we can do that by describing either the characters that were modified (syntactic approach), or by describing the changes to the models in terms of the meaning of the model; in our case, in terms of configurations.

Let us consider the small variability model below, described in the Kconfig model. It contains 3 features, A, B, and C. The possible configurations are obviously: {A}, {A,B}, {A,C}, and {A,B,C}.

```
config A "my root feature"
default 1

config B "sub feature 1"
depends on A

config C "sub feature 2"
depends on A
```

If a developer adds a new feature D relying on A to that model, under the root, we obtain the following variability model:

```
config A "my root feature"
default 1

config B "sub feature 1"
depends on A

config C "sub feature 2"
depends on A
```

```
config D "new feature"
depends on A
```

The possible configurations described by this model are : {A,B}, {A,D}, {A,C}, {A,B,C}, {A,B,D}, {A,C,D}, and {A,B,C,D}.

We can describe this change syntactically by providing the following information:

```
the change added the following string:
---
config D "new feature, just added by a dev."
depends on A
---
to the file describing the variability model
```

This allows us to describe what transformation took place with respect to the textual representation of our variability model.

Alternatively, we can consider that the variability model is a representation, and the change is, in fact, a change to the valid set of configuration of our system. In that sense, we could describe the change in a "semantic" manner, based on the meaning of model, as follows:

```
the change added the following set of valid configurations
---
{A,D},{A,B,D}, {A,C,D}, and {A,B,C,D}
---
to the file describing the variability model
```

This gives immediate information regarding the context in which the new feature D may be used. It indicates that it can rely on "A" (which was known already thanks to the textual differencing approach). It also tells us that it should be able to run along side of feature B and (or) C.

Syntactic and semantic differencing provide different information. Syntactic differencing is useful to know what changes are performed by developers on the different artefacts. Semantic differencing, on the other hand, gives us better information on the consequences of that change on the system, but hides the details of why such changes occurred.

The most notable difference between syntactic and semantic differencing approaches is the output. Syntactic differencing approaches provide the description of a change expressed using new concepts. In our example, the output of a syntactic differencing is the

textual change that occurred. That change is not in itself a feature, nor a configuration - it is the description of a textual change within a file. On the contrary, semantic differencing provides an output which is of the same nature of its inputs(Binkley et al., 2001; Maoz et al., 2011a). In our example, semantic differencing provides a set of configurations as an output. This set can also be represented as a variability model. This makes syntactic changes harder to integrate back into the model, or apply the change provided by the differencing process to another model. In a more general context, semantic differencing is not limited to variability models. Similar approaches were used on other types of models such as UML class diagrams (Maoz et al., 2011b) or UML activity diagrams (Maoz et al., 2011a).

9.2. SYNTACTIC AND SEMANTIC VARIABILITY MODEL CHANGES IN LINUX

The FMDiff approach is fundamentally a syntactic approach. We rely on a semi-structured diff' methodology to identify which attributes of features are affected by a change. As a result, a change to a single feature within a Kconfig file will result in the creation of a FMDiff "change". The information we capture is extracted from the changes performed by developers of that feature. Therefore, FMDiff reports textual changes, as performed by developers, on the features of the system within Kconfig files: changes to the syntax contained within the file. This gives us insights on which entity was targeted by the change and how the feature attributes and its relationships evolved.

There is a different way of describing changes to a variability model (VM). Because VMs are compact representations of the valid configurations of a system, a change to a feature yields changes to the set of valid configurations. A semantic differencing approach would provide change information in terms of modified configurations: changes to the meaning of the feature and their constraints. The set of added and removed configurations following a change can itself be represented using feature models, since they are sets of configurations. The semantic description of a change in a VM should be representable as a VM itself.

9.3. LIMITATIONS OF SYNTACTIC DIFFING ON THE LINUX KERNEL

As mentioned earlier in this chapter, in order to extract changes occurring on the Linux VM, we use a methodology which requires the selection of an architecture. Once the architecture is selected, the process rebuilds a VM corresponding to the features "included" in that architecture.

In their work on the Linux kernel, Rothberg et al. showed how this inclusion of certain features in an architecture-specific VM was a purely syntactical inclusion. This appeared to be true for a number of features, but more specifically relevant for features representing drivers. They did so by showing that such features that "included" in a architecture-specific VM are actually not "selectable". In that sense, the feature is included in the model, but existing constraints prevent it from being included in a valid configuration. Therefore, the feature is present in the architecture-specific VM on a syntactic level, but since it cannot be included in a configuration it has no consequence on the "semantics" of the model.

In our study, we compared the evolution of the different architecture-specific VM that we could obtain from the kernel. This allowed us to show that changes mostly oc-

curred in either one architecture-specific VM, or all architecture-specific VMs. Our work allowed us to establish that the specific architecture-specific VMs actually evolved differently.

In light of the work of Rothberg et al., it is clear that some changes that we observed as being shared by all architecture-specific VMs were in fact performed on features that were not necessarily selected within that architecture. Therefore, drawing the conclusion that changes of such features participate in the evolution of the architecture-specific VM would be fallacious. Rothberg et al. showed that the different architecture-specific VM had evolutions that were much more dissimilar than what we established with our work on FMDiff.

Rothberg's approach differs from ours in that it filters the features contained in each architecture-specific VM before considering changes. Beyond that point, the identification of feature changes remains quite comparable, both being based on textual identification of changed features. In that sense, Rothberg approach's is not applying semantic differencing per se, but improving on our syntactic approach by adding selectability information.

9.4. CHALLENGES

Semantic differencing, especially in the context of variability modelling, seems to be a promising direction (Fahrenberg et al., 2011) but comes with a number of challenges. Performing "semantic differencing" on a feature model should result in lists of configurations - since a feature model is a compact representation of allowed configurations, semantic changes should be described in terms of configurations as well. Following a feature change, one should produce a representation of configurations that were not available before and now are, and vice-versa. However, obtaining the semantic difference between two variability models and then exploiting this information can be difficult.

In the Linux kernel, the variability model contains more than 13,000 features. The number of configurations impacted by a change might be very large. Recomputing the whole set of possible configurations, before and after the change, to identify changes, will be computationally very intensive. Note that computing all possible configurations of a simple feature model requires an exponential computation time, based on the number of features. Therefore, with 13,000 features or more, the computation will take a lot of time. If we consider that the Linux tree includes roughly 10,000 patches per release and a release lasts for 6 weeks [18], then the kernel receives approximately 238 patches per day (approximately one patch every 6 minutes). There is little chance of being able to compute, the results of semantic differencing on every patch, let alone make use of it.

Then, once the changes configurations have been identified, each of them may contain any number of features which depends on the number of features available in the system. This yields another issue: how to display changes of configurations in such a way that developers are not flooded by the information? A list of tens, or hundreds of added or removed configurations, each containing hundreds of features will be difficult to understand by maintainers. Therefore, in order to be useful for change comprehension, we need to find insightful means to present such data. While the idea to build a variabil-

[18] http://arstechnica.com/information-technology/2015/02/linux-has-2000-new-developers-and-gets-10000-patches-for-each-version/

ity model representation for such changes is enticing, we should point out that reverse engineering of variability models from configurations does not necessarily provide the most human-readable models (Becan, 2013; She et al., 2011).

If we can produce such change descriptions, we have to determine how they can be of use for engineering tasks beyond change comprehension. This could be done if we consider that integrating a change described as a VM to another VM is a composition operation. Composing models, and feature models in particular, can be challenging in itself (Acher et al., 2010a,b).

Each of those challenges constitutes, in our opinion, a research topic on its own. We did not attempt to tackle them. While less precise than semantic differencing, the FMDiff approach was able to shed new light on the evolution of the kernel VM. It allowed us to identify changed features, and draw conclusions on the types of changes that are prevalent in such context. And while we underestimated this effect, we showed that architecture-specific VMs have different evolution paths, and one cannot take a single one and extrapolate observations on the entire kernel.

3

Evaluating Feature Change Impact on Multi-Product Line Configurations Using Partial Information

The beauty of the universe consists not only of unity in variety, but also of variety in unity.

Umberto Eco - *The Name of the Rose*

Evolving large-scale, complex and highly variable systems is known to be a difficult task, where a single change can ripple through various parts of the system with potentially undesirable effects. In the case of product lines, and moreover multi-product lines, a change may affect only certain variants or certain combinations of features, making the evaluation of change effects more difficult.

In this paper, we present an approach for computing the impact of a feature change on the existing configurations of a multi-product line, using partial information regarding constraints between feature models. Our approach identifies the configurations that can no longer be derived in each individual feature model taking into account feature change impact propagation across feature models. We demonstrate our approach using an industrial problem and show that correct results can be obtained even with partial information. We also provide the tool we built for this purpose.

This chapter was originally published as "Evaluating Feature Change Impact on Multi-Product Line Configurations Using Partial Information" in Proceedings of the 14th Conference on Software Reuse (ICSR'15) authored by Dintzner, Kulesza, Van Deursen and Pinzger.

1. INTRODUCTION

Evolving large-scale, complex and variable systems is known to be a difficult task, where a single change can ripple through various parts of the system with potentially undesirable effects. If the components of this system are themselves variable, or if the capabilities exposed by an interface depend on some external constraint (i.e., configuration option), then engineers need extensive domain knowledge on configuration options and component implementations to safely improve their system (Heider et al., 2012b). In the domain of product line engineering (PLE), an approach aiming at maximising asset reuse in different products (Pohl et al., 2005), this type of evolutionary challenge is the norm. Researchers and practitioners have looked into what variability modeling - and feature modeling specifically - can bring to change impact analysis on product lines (PLs). Existing methods can evaluate, given a change expressed in features, how a feature model (FM) and the composition of features it allows (configurations) are impacted (Heider et al., 2012a; Paskevicius et al., 2012; Thuem et al., 2009). However, FMs grow over time in terms of number of features and constraints and safe manual updates become unmanageable by humans (Bagheri and Gasevic, 2011). Moreover, automated analysis methods do not scale well when the number of configurations or feature increases (Heider et al., 2012a).

To mitigate this, *nested product lines*, *product populations*, or *multi-product lines* (MPL - a set of interdependent PLs) approaches recommend modularizing FMs into smaller and more manageable pieces (Krueger, 2006; Ommering and Bosch, 2002; Schröter et al., 2013). While this solves part of the problem, known FM analysis methods are designed for single FMs. A common approach is to recompose the FMs into a single one. To achieve this, existing approaches suggest describing explicitly dependencies between FMs using cross-FM constraints, or hierarchies (Acher et al., 2010a) to facilitate model composition and analysis. Such relationships act as vectors of potential change impact propagation between FMs. However, Holl et al. noted that the knowledge of domain experts about model constraints is likely to be only partial (both intra-FMs or extra-FMs (Holl et al., 2012)). For this reason, we cannot assume that such relationships will be available as inputs to a change impact analysis.

In this context, we present and evaluate an approach to facilitate the assessment of the impact of a feature change on existing configurations of the different PLs of an MPL using partial information about inter-FMs relationships. After giving background information regarding feature modelling and product lines (Section 2), we present the industrial problem that motivated this work and detail the goals and constraints of this study (Section 3). We then present our approach to enrich the variability model of an MPL using existing configurations of individual FMs, and the heuristic we apply when analyzing the effect of a feature change on existing configurations of an MPL (Section 3). In Section 5, we assess our approach in an industrial context. We present and discuss how we built the appropriate models, the output of our prototype implementation and the performance of the approach with its limitations. Finally, Section 9 presents related work and we elaborate on possible future work in Section 10.

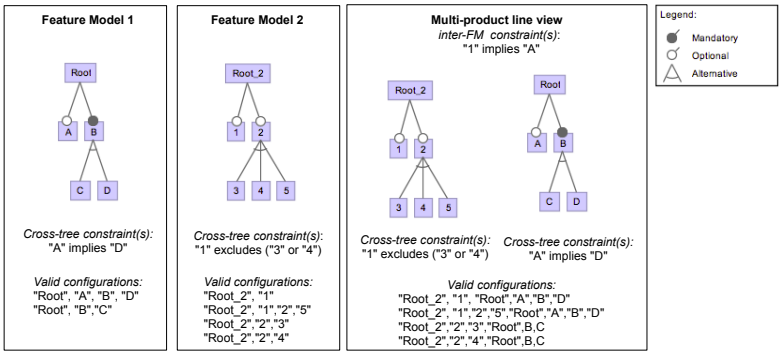

Figure 3.1: Example of FMs in a SPL and MPL context

2. BACKGROUND

In this paper, the definition of *feature* given by Czarnecki et al. in (Czarnecki et al., 2005) is used: "a feature may denote any functional or nonfunctional characteristic at the requirement, architectural, component, platform or any other level". A feature model (FM) is a structured set of features with selection rules specifying the allowed combinations of features. This is achieved through relationships (optional, mandatory, part of an alternative or OR-type structures) and cross-tree constraints - arbitrary conditions on feature selection. The most common types of cross-tree constraints are "excludes" (e.g., "feature A excludes feature B") and "implies" (Kang et al., 1990). With a FM, one can derive configurations: a set of features which does not violate constraints established by the FM. An example of simple FMs with their valid configurations are depicted on the left hand side of Figure 3.1.

In the context of a multi-product line, several inter-related FMs are used to describe the variability of a single large system. This can be achieved by creating "cross-feature model" constraints or through feature references (Acher et al., 2010c) - where a given feature appears in multiple FMs. The constraints between FMs can be combination rules referring to features contained within different models. Those constraints can also be derived from the hierarchy (or any imposed structure (Reiser and Weber, 2007), (Acher et al., 2010c)) of the FMs involved in an MPL. In those cases, the combination rules can refer to both features and FMs. A product configuration derived from an MPL is a set of features which does not violate any constraints of individual FMs nor the cross-FM constraints that have been put in place. An example of combined FMs with a constraint between two FMs can be seen on the right hand side of Figure 3.1.

3. MOTIVATION: CHANGE IMPACT IN AN INDUSTRIAL CONTEXT

Our industrial partner builds and maintains high-end medical devices, among which an x-ray machine. This x-ray machine comes in many variants, each differing in terms of hardware (e.g., tables, mechanical arms) and software (e.g., firmware version, imaging system). Certified third party products can be integrated through different types of external interfaces: mechanical (e.g., a module placed on the operating table), electrical

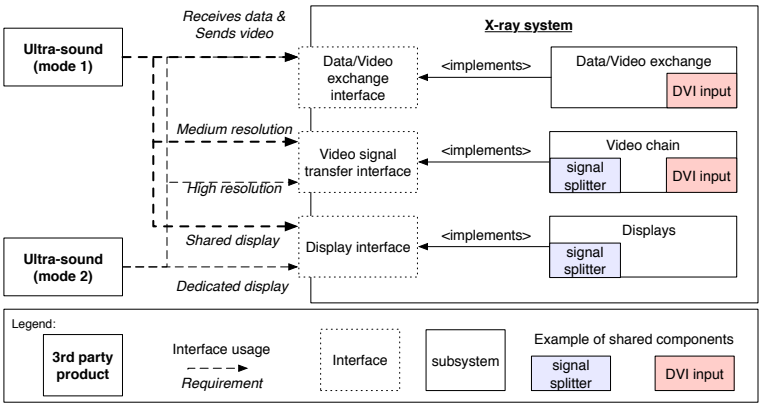

Figure 3.2: X-ray machine system overview

(inbuilt power supply), data related (image transfer). As an example, three main subsystems of the x-ray machine (data/video exchange, video chain, and display) and three main interfaces (display interface, video signal, and data/video exchange) are shown in Figure 3.2. The two working modes of a given 3rd party product ("mode 1" and "mode 2") use the same interfaces in slightly different ways. In "mode 1", the 3rd party product reuses the x-ray machine display to show images ("shared display") while in "mode 2" a dedicated display is used. Sharing an existing display implies using a signal splitter/merger in the display subsystem. But the splitter/merger also plays a role in the video processing chain and is only available in certain of its variants.

Following any update, engineers must validate if the new version of the system can still provide what is necessary for 3rd party product integration. This leads to the following type of questions: *"Knowing that 3rd party product X uses the video interface to export high resolution pictures and import patient data, is X supported by the new version of the x-ray machine?"*. Let us consider the following scenario: a connection box, present in the video chain and data/video exchange subsystems, is removed from the list of available hardware. Some specific configurations of the video chain and of the data/video exchange subsystems can no longer be produced. The data/video exchange interface required the removed configurations to provide specific capabilities. Following this, it is no longer possible to export video and import data and the integration with the 3rd party product is compromised.

Currently, engineers validate changes manually by checking specification documents (either 3rd party products requirements or subsystem technical specifications) and rigorous testing practices. Despite this, it remains difficult to assess which subsystem(s) and which of their variant(s) or composition of variants will be influenced by a given change. Given the rapid evolution of their products, this error-prone validation is increasingly time consuming. Our partner is exploring model-driven approaches enabling early detection of such errors.

While this example is focused on the problems that our industrial partner is facing, enabling analysis for very large PLs and MPLs is a key issue for many companies.

Recently, Schmid introduced the notion of variability-rich eco systems (Schmid, 2013), highlighting the many sources of variability that may influence a software product. This further emphasizes the need for change impact analysis approaches on highly variable systems.

4. FEATURE-CHANGE IMPACT COMPUTATION

Given the problem described in the previous section, we present here the approach we designed to assist domain engineers in evaluating the impact of a change on their products. We first describe the main goal of our approach and our contributions. Then, we detail the approach and illustrate it with a simple example. Finally, we consider the scalability aspects of the approach and present our prototype implementation.

4.1. GOALS AND CONSTRAINTS

For our industrial partner, the main aim is to obtain insights on the potential impacts of an update on external interfaces used by 3rd party products. However, we have to take into account that domain engineers do not know the details of the interactions of the major subsystems (Holl et al., 2012) nor all components included in each one - only the ones relevant to support external interfaces. As an input, we rely on the specifications of each major subsystem and their main components in isolation as well as their existing configurations. Because of the large number of subsystem variants and interface usages (choices of capabilities or options), we consider each of them as a product line (PL) in its own right. Features then represent hardware components, (non-)functional properties, software elements, or any other relevant characteristic of a subsystem or interface. Using a simple feature notation and cross-tree constraints (Kang et al., 1990), we formalize within each subsystem the known features and constraints between them. By combining those PLs, we obtain a multi-product line (MPL) representation of the variability of the system.

With such representation, a change to a subsystem or interface can be expressed in terms of features: adding or removing features, adding, removing or modifying intra-FM constraints. Once the change is known, we can apply it to the relevant FM and evaluate if existing configurations are affected (no longer valid with respect to the FM). Then, we determine how the change propagates across the FMs of the MPL using a simple heuristic on configuration composition. As an ouput, we provide a tree of configuration changes, where nodes are impacted FMs with their invalid configurations.

Our work brings the following main contributions. We present a novel approach to compute feature change impact on existing configurations of an MPL. We provide a prototype tool supporting our approach, available for download.[1] We demonstrate the applicability of the approach by applying it to a concrete case-study executed in cooperation with our industrial partner.

4.2. APPROACH

We describe here first how the model is built. Then, we show how we enrich the model with inferred information and finally the steps taken for simulating the effects of a fea-

[1] The tool is available at `https://bitbucket.org/NJRD/mpl-change-impact`

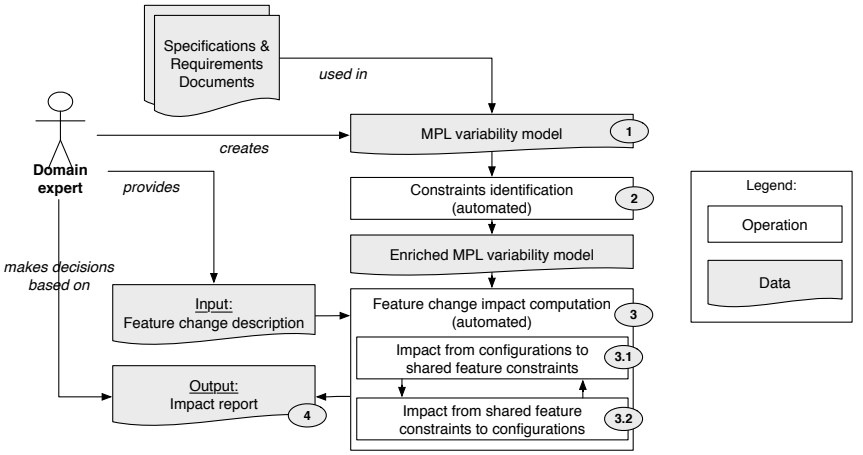

Figure 3.3: Approach overview

ture change on existing configurations. An overview of the different steps are shown in Figure 3.3.

Step 1: Describe subsystem variability. The first step of our approach consists in modelling the various subsystems using FM notation. This operation is done by domain experts, using existing documentation. When a subsystem uses a feature that has already been described in another subsystem, we reference it instead of creating a new one (Acher et al., 2010a). Such features are considered as "shared" between the different FMs. We associate with each FM its known configurations.

Step 2: Enrich the model with inferred composition rules. Once all FMs and configurations have been described, we use the configurations to infer how pairs of subsystems can be combined. We identify, in FMs sharing features, which features are shared and then create a list of existing partial configurations containing only them. To do so, we rely on feature names to identify which features appear in more than one FM. Partial configurations appearing in existing configurations of both FMs constitute the whitelist of partial configurations enabling composition of configurations between the involved FMs. For two given FMs, the number of feature involved in shared feature constraints is equal to the number of features shared between them. Those partial configurations are the *shared feature constraints* relating pairs of FMs: two configurations, from two different FMs sharing features, are "compatible" if they contain exactly the same shared features. In order to apply such heuristic, shared feature constraints must be generated between every pairs of FMs sharing features. An example of such constraints is shown in Figure 3.4, where FMs 1 and 2 share features E and D.

Step 3: Compute the impact of a feature change. We use the enriched model to perform feature change impact computation at the request of domain experts. A feature change can be any modification of a FM (add/remove/move/modify features and constraints) or a change in available configurations (add/remove). We assess the impact of the change of the configurations of a modified FM by re-validating them with respect to the updated FM, as suggested in (Heider et al., 2012a). This gives us a first set of invalid

configurations that we use as a starting point for the propagation heuristic.

Step 3.1: Compute impact of configuration changes on shared feature constraints. We evaluate how a change of configuration of a FM affects the shared feature constraints attached to it. If a given shared feature constraint is not satisfied by at least one configuration of the FM then it is invalidated by the change. For each FM affected by a configuration change, we apply the reasoning presented in Algorithm 1. In the case a change does not modify existing configurations, this step will tell us that all existing constraints are still valid, but some can be added. Otherwise, if all configurations matching a constraint have been removed then that constraint is considered invalid (i.e., does not match a possible combination of configurations). Given a list of invalid shared feature constraints and the FMs to which it refers to, we can execute Step 3.2. If no shared feature constraints are modified, the computation stops here.

Data: a FM fm with an updated set of configurations
Result: a list of invalidated shared feature constraints $lInvalidConstraints$

foreach *shared feature constraint* sfc *in* fm **do**
  $allowedFeatures \leftarrow$ selected features of sfc;
  $forbiddenFeatures \leftarrow$ negated features of sfc;
  foreach *configuration* c *in* fm **do**
    if $allowedFeatures \subset c$ **then**
      if $c \cap forbiddenFeatures == \emptyset$ **then**
        c is compliant;
      end
    end
  end
  if *no compliant configuration found* **then**
    add sfc to $lInvalidConstraints$;
  end
end

Algorithm 1: Configuration change propagation

Step 3.2: Compute impact of shared feature constraint changes on configurations. Given a set of invalid shared feature constraints obtained in the previous step, we evaluate how this invalidates other FMs configurations. If a configuration of an FM does not match any of the remaining shared feature constraints, it can no longer be combined with configurations of other FMs and is considered invalid. We apply the operations described in Algorithm 2. If any configuration is invalidated, we use the output of this step to re-apply Step 3.1.

Step 4: Consolidate results. We capture the result of the computation as a tree of changes. The first level of the tree is always a set of configuration changes. If more than one FM is touched by the initial change (e.g., removal of a shared feature) then we have a multi-root tree. Each configuration change object describes the addition or removal of any number of configurations. If a configuration change triggered a change in shared feature constraints, a shared feature constraint change is added as its child. A shared fea-

```
Data: fm: a FM with updated shared feature constraints
Result: lInvalidConfs: a list of invalidated configurations of fm
foreach configuration c in fm do
    foreach shared feature constraint sfc in fm do
        allowedFeatures ← selected features of sfc;
        forbiddenFeatures ← negated features of sfc;
        if allowedFeatures ⊂ c then
            if c ∩ forbiddenFeatures == ∅ then
                c is compliant;
            end
        end
    end
    if no compliant constraint found then
        add c to lInvalidConfs;
    end
end
```

Algorithm 2: Shared feature constraint change propagation

ture constraint change references the two FMs involved and any number of constraints that were added or removed. The configuration changes following this shared feature constraint modification are then added as a child "configuration change object". This structure allows us to describe the path taken by the impact propagation through the different FMs.

4.3. EXAMPLE

Let us consider the example shown in Figure 3.4, where two FMs share two features: D and E. The model is enriched with the "shared feature constraints" deduced from existing configurations. Those constraints state that, for a configuration of FM1 and FM2 to be combined, both of them need to have shared features that are either (E,D), (D, not E) and (not E, not D). The resulting data structure is shown on the left hand side of Figure 3.4.

We consider the following change: Configuration 1.2 is removed, operation marked as 1 in Figure 3.4. We apply the algorithm described in Step 3.1, using FM1 as a input, and with Configurations 1.1 and 1.3 (all of them except the removed one) and the associated 3 shared feature constraints. For Constraint 1, the allowed features are "E" and "D", and there are no forbidden features. We search for existing configurations of FM1 containing both "E" and "D" among Configurations 1.1 and 1.3. We find that Configuration 1.3 satisfies this constraint. The Constraint 2 (allowing "D" and forbidding "E") is not matched by any configurations, since the only configuration containing "D" and not "E" is Configuration 1.2 has been removed. Constraint 3 forbidding features "D" and "E" is satisfied by Configuration 1.1. The resulting list of invalid constraints contains only one element: Constraint 2 (marked as operation 2 in the diagram).

We then apply 2 presented in Step 3.2 to assess the effect of that change on the con-

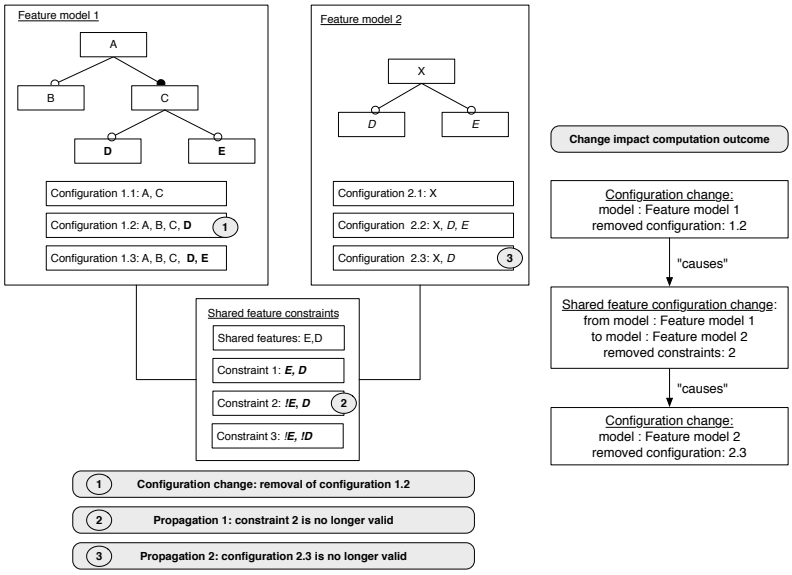

Figure 3.4: Change impact propagation example

figurations of other FMs (FM2 only in this case). With the remaining Constraints 1 and 3, we run through the configurations of FM2 to identify which configurations no longer satisfy any constraints. We find that Configuration 2.1 satisfies Constraint 3 (does not contain "D" nor "E"), and Configuration 2.2 satisfies Constraint 1 (contains both "E" and "D"). However, configuration 2.3 does not satisfy any of the remaining constraints and for this reason, is marked as invalid (shown as operation 3 on the diagram).

On the right hand side of Figure 3.4, we present the resulting tree (a branch in this case). The intial change (removal of configuration 1.2 of FM1) is captured by the first "configuration change" object. Changes to shared features constraints are directly attached to this configuration change: the "shared feature configuration change" object. Finally, the last node of the tree is the invalidation of Configuration 2.3 of FM2.

4.4. SCALABILITY ASPECTS

The initial step of our approach replicates what Heider suggests in (Heider et al., 2012a): reinstantiating existing configurations. Such approaches are known as *product-based* approaches (Thüm et al., 2014a). They have known drawbacks: as the number of configurations and features increases, the solution does not scale. By placing ourselves in an MPL environment, we have small to medium size FMs to analyze and perform this type of operation only on individual FMs.

Our composition heuristic focuses on composition of configurations (as opposed to composition of FMs). Once the local product-based approach is used, we rely on it to identify broken compositions of configurations across the FMs without having to revalidate any configurations against the FMs. This last step can be viewed as a *family-based* analysis of our product line (Thüm et al., 2014a), where we validate a property over all

members of a PL. We store information relative to shared feature constraints on the model itself. With this information, applying the heuristic to an MPL amounts to searching specific character strings in an array, which is much faster than merging models or validating complete configurations.

4.5. PROTOTYPE IMPLEMENTATION

We implemented a prototype allowing us to import FMs into a database, enrich the model and run feature change impact computations. The choice of using a database was motivated by potential integration with other datasources. Since FMs are mostly hierarchical structures, we use Neo4j.[2] Our Neo4j schema describes the concepts of feature model, feature, configuration and shared feature constraint with their relationships as described in the previous section. This representation is very similar to other FM representations such as (Thüm et al., 2014b) with one exception. The mandatory, optional or alternative nature of a feature is determined by its relationship with its parent; as opposed to be a characteristic of the feature itself. This allows to have an optional feature in a FM, referenced by another FM as part of an alternative.

We leverage the Neo4j *Cypher* query language to retrieve relevant data: shared features, configurations containing certain features as well as interconnected feature models and the features which links them. We use FeatureIDE (Thüm et al., 2014b) as a feature model editor tool. We import models in their xml format into our database using a custom java application. A basic user interface allows us to give the name of a feature to remove, run the simulation, and view the result.

5. INDUSTRIAL CASE STUDY

As mentioned in Section 3, this paper is motivated by an industrial case study proposed by our partner. The end-goal of this case study is to assess the applicability of our approach in an industrial context. To do so, we reproduce a past situation where a change modified the behaviour of some products of their product line on which a 3rd party product was relying, and where the impact was detected late in the development process. We present and discuss the main steps of our approach and their limitations when applied in an industrial context: the construction of the model, the feature change impact computation with its result, and the performance of our prototype implementation.

5.1. MODELLING A X-RAY MPL

We start by gathering specification documents of the main subsystems identified in Section 3, as well as 3rd party product compatibility specifications. With the domain experts, we identify relevant components and existing configurations of each subsystem. Using this information, we model the three interfaces and three subsystems presented in Figure 3.2 as six distinct feature models (FMs). The three interfaces are (i) the video/data transfer interface (data and video transfer capabilities), (ii) the video export interface specifying possible resolutions and refresh rates, and finally (iii) the display interface representing a choice in monitor and display modes. 3rd party product interface usages are modeled as the configurations associated to those FMs. The three subsystems

[2] http://www.neo4j.org

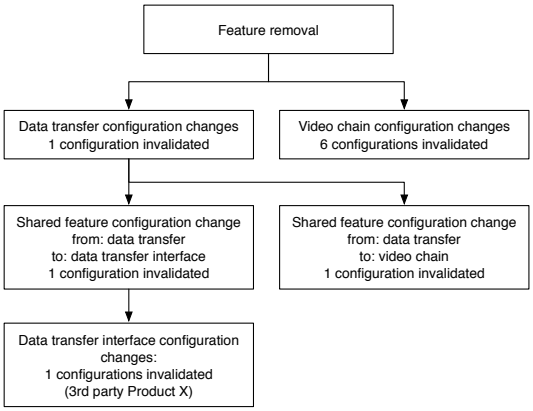

Figure 3.5: Output of the feature removal simulation

of the x-ray machine are (i) the data/video transfer subsystem, (ii) the video chain used to transport images from a source to a display, and finally (iii) display subsystem. Configurations of those subsystems are the concrete products available to customers (between 4 and 11 configurations per FM). Each FM contains between 10 and 25 features, with at most 5 cross-tree constraints. The “data transfer”, “video chain”, and “screen” FMs share features relating to hardware components, and reuse features from interface FMs. We use FeatureIDE to create FMs and configurations. We then import them into a Neo4J database and use our prototype implementation to generate the necessary shared feature constraints as described in Section 3.

The main challenge of this phase is to ensure that shared features represent the same concept in all FMs. For instance, a feature “cable” refers to one specific cable, in a specific context, and must be understood as such in all FMs including it. Misreferencing features will lead to incorrect shared feature constraints and incorrect change impact analysis results. We mitigated this effect by carefully reviewing FMs and shared features with domain expert.

5.2. SIMULATING THE CHANGE

We studied the effect of the removal of a hardware component used to import video into the system. To simulate this with our prototype, we provide our tool with the name of the feature to remove (“Connection box 1”). “Connection box 1” is included in both the “data/video transfer” and “video chain” FMs, so its removal directly impacts those two FMs. The tool re-instantiates all configurations of those two FMs and find that 6 configurations of the “video chain” FM, and 1 from the “data transfer” FM are invalid. Then, the prototype executes the propagation heuristic. A shared feature constraint between the “data transfer” and “data transfer interface” FMs is no longer satisfied by any configuration of the “data transfer” FM, and is now invalid. Without this shared feature constraint, one configuration of the “data transfer interface” FM can no longer be combined with the “data transfer” FM and is considered as invalid. The removal of a configuration in an interface FM tells us that the compatibility with one 3rd party product is no longer pos-

sible. The modifications of the "data transfer" FM also invalidated a shared feature constraint existing between the "data transfer" and "video chain" FMs. However, the change of "shared feature constraint" did not propagate further; the configurations that it should have impacted had already been invalidated by a previous change.

The result of this impact analysis, reviewed with 3 domain experts, showed the impact of interfaces that had been problematic in the past. We ran several other simulations on this model (removal of features, removal of configurations). On each occasion, the result matched the expectations of domain experts - given the data included in the model. In this context, the approach proved to be both simple and successful. This being said, by using information from existing configurations, we over-constrain inter-FMs relationships. If a shared optional feature is present in all configurations of a given FM, it will be seen as mandatory during impact computation. However, if a feature is present in all of existing configurations, it is mandatory with respect to what is available - as opposed to mandatory in the variability model. As long as we reason about existing configurations only, using inferred shared feature constraints should not influence negatively the result of the simulation.

5.3. PERFORMANCE ANALYSIS

We provide here a qualitative overview of performance measurements that were performed during this case study. For our main scenario, our approach checked all configurations of 2 of the FMs, and the change propagated to 2 others. 2 of the 6 FMs did not have to be analyzed. In this specific context, our implementation provided results in less than a few seconds, regardless of the scenario that was ran. We then artificially increased the size of the models (number of features and number of configurations) to evaluate how it influences the computation time of the propagation algorithm. Given a set of invalid configurations, we measure how long it takes to assess the impact on one connected FM. For 2 FMs with 20 features each and 20 configurations each, sharing 2 features, the propagation from 1 FM to the other and impact its configurations takes approximately 450ms. With 200 configurations in each FMs, the same operation takes 1.5s; and up to 2.5s for 300 configurations.

During the industrial case study, the performance of the prototype tool was sufficient to provide almost real-time feedback to domain engineers. The size of the models and the number of configurations affect negatively the computation time of the change impact analysis, because the first step of our approach is product-based: we do check all configurations of the initially impacted FMs. However, using an MPL approach, individual FMs are meant, by design, to be relatively small. Then, computing the propagation of those changes, if any, depends on the number of affected FMs as defined by our propagation heuristic. The heuristic itself is the validation of a property over all members of the product family ("family-based" approach), so its performance is less influenced by model size (Thüm et al., 2014a). This operation consists in searching for strings in an array, which should remain manageable even for large models. Our naive implementation, using Neo4j, already provided satisfactory performance.

5.4. THREATS TO VALIDITY

With respect to *internal validity*, the main threat relates to the construction of the models used for the industrial case study. We built the different FMs and configurations of the case study using existing documentation while devising the approach. To avoid any bias in the model construction, we reviewed the models several times with domain experts, ensuring their representativeness.

Threats to *external validity* concern the generalisation of our findings. For this study, we used only the most basic FM notation (Kang et al., 1990). Our approach should be applicable using more complex notations as long as those notation do not change the representation of the configurations (list of feature names, where each name appear once). If, for instance, we use a cardinality-based notation, the heuristic will have to be adapted to take this cardinality into account. The extracted information from existing configurations was sufficient for this case study, but more complex relationships between FMs might not have been encountered. Applying our approach on a different PL would confirm or infirm this.

6. RELATED WORK

The representation of variability in very large systems, using multiple FMs, has been studied extensively during the past few years. Several composition techniques have been devised. Composition rules can be defined at an FM level, specifying how the models should be recombined for analysis. Otherwise, cross-FM constraints can be defined. Examples can be found in the work of Schirmeier (Schirmeier and Spinczyk, 2009) and Acher (Acher et al., 2010a,b). In our context, we chose not to follow those approaches as we do not know a priori the over-arching relationships between FMs, nor can we define cross-FM constraints since we work with partial information. Moreover, those techniques would then require us to re-compose models before validating the various configurations which, as noted in (Hartmann and Trew, 2008), is complex to automate. Recent work on MPLs showed that there is a need to specialise feature models to segregate concerns in MPL variability models. Reiser et al. (Reiser and Weber, 2007) propose the concept of "context variability model" for multi-product lines, which describes the variability of the environment in which the end product resides. In our study, we classified our FMs as either interface or subsystem. This classification also allows us to qualify the configurations (as interface usage or product implementation), which proved to be sufficient for our application. Schröter et al. present the idea of interface FMs where specific FMs involved in an MPL act as interfaces between other FMs (Schröter et al., 2013). They propose a classification of the characteristics that can be captured by such models (syntactic, behavioral, and non-functional). While we did not use this approach directly, we noted that for non-interface FMs, we used specific branches of the model to organize reused shared features. It is interesting to note that the designs of (non-interface) FMs share a common structure. We used specific branches of their respective FM to organise features shared with interface FMs. Doing so, we specialized a branch of a FM instead of creating dedicated FMs and we do not restrict the type of features it contains (functional and non-functional alike).

Heider et al. proposed to assess the effect of a change on a variability model by re-instantiating previously configured products (Heider et al., 2012a), and thus validating

non-regression. Our approach applies similar principles, as we will consider a change safe as long as the existing products can be re-derived. We apply those concepts in a multi-product line environment, where change propagation is paramount. Thüm et al. (Thuem et al., 2009) proposed to classify changes occurring on feature models based on their effect on existing product configurations. The change is considered as a "generalisation" if the set of valid products has been extended, "specialisation" if it has been reduced, "refactoring" if it has not changed, and "arbitrary edit" in all other cases (when some configurations were removed and others added). This initial classification gave us some insight into the potential impact of a change, but only for a single FM. Their methodology could be applied during the initial step of our approach to identify changes that do not affect existing configurations, avoiding extra computation later on.

7. CONCLUSION

Understanding the full extent of the impact of a change on a complex and highly variable product is a difficult task. The main goal of this research is to facilitate the evolution of such systems by assisting domain experts in assessing the effects of changes on multi-product line variability models. In this paper, we presented an approach to compute the impact of a feature change on a multi-product line for non-regression purposes, leveraging information contained in existing product configurations to infer feature model composition constraints. We described how our modelling approach can be used in a practical context, using an industrial case and provide a qualitative review of the performance of our prototype tool. With partial information, we were able to accurately identify which configurations of an MPL were rendered invalid by a feature change.

As industrial products grow more complex and become more variable, managing their evolution becomes increasingly difficult. Approaches supporting domain experts' activities will have to be adapted to meet new challenges. As a step in that direction, we released our implementation as an open source project [3] as well as the dataset we used for the performance evaluation. We then plan to integrate it into existing feature modelling tools. We intend to explore how we can make the best use of the promising graph database technologies such as Neo4J for feature model checking. With such technology, we will be in a position to consider more complex models, with potentially more complex FM composition constraints, further facilitating the design, analysis and maintenance of highly variable systems.

[3] The tool is available at `http://swerl.tudelft.nl/bin/view/NicolasDintzner/WebHome`

4

FEVER: Feature-oriented Changes and Artefact Co-evolution in Highly Configurable Systems

The only way to make sense out of change is to plunge into it, move with it, and join the dance.

Alan W. Watts

The evolution of highly configurable systems is known to be a challenging task. Thorough understanding of configuration options their relationships, and their implementation in various types of artefacts (variability model, mapping, and implementation) is required to avoid compilation errors, invalid products, or dead code.

Recent studies focusing on co-evolution of artefacts detailed feature-oriented change scenarios, describing how related artefacts might change over time. However, relying on manual analysis of commits, such work do not provide the means to obtain quantitative information on the frequency of described scenarios nor information on the exhaustiveness of the presented scenarios for the evolution of a large scale system.

In this work, 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 apply this methodology to the Linux kernel and build a dataset comprised of 15 releases of the kernel history.

We performed an evaluation of the FEVER approach by manually inspecting the data and compared it with commits in the system's history. The evaluation shows that FEVER accurately captures feature related changes for more than 85% of the 810 manually inspected

commits. We use the collected data to reflect on occurrences of co-evolution in practice. Our analysis shows that complex co-evolution scenarios occur in every studied release but are not among the most frequent change scenarios, as they only occur for 8 to 13% of the evolving features. Moreover, only a minority of developers working on a given release will make changes to all artefacts related to a feature (between 10% and 13% of authors).

While our conclusions are derived from observations on the evolution of the Linux kernel, we believe that they may have implications for tool developers as well as guide further research in the field of co-evolution of artefacts.

This chapter was originally submitted as “FEVER: Feature-oriented Changes and Artefact Co-evolution in Highly Configurable Systems” to the journal of "Empirical Software Engineering" (submitted in October 2016, currently under review) authored by Dintzner, Van Deursen and Pinzger. An earlier version of this work appeared in the proceedings of the 13th International Conference on Mining Software Repositories (MSR'16), authored by Dintzner, Van Deursen and Pinzger

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(van Gurp et al., 2001). The benefits of such development strategies are to reduce the time to market, as mass-customization facilitates the creation of tailored solutions, and improved software quality, as re-used components are tested in various contexts (Clements and Northorp, 2002). Examples of such systems can be found in various domains, such as database management (Batory et al., 1988; Rosenmüller et al., 2008), SOA based systems (Kumara et al., 2013), operating systems (Berger et al., 2010), and a number[1] of industrial and open source software projects (Liebig et al., 2010) among which the Linux kernel may be the best known.

A constraint of such a development strategy is the fragmentation of concerns among development artefacts in such a way that re-use and customization can be achieved. 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 (Dietrich et al., 2012b; Neves et al., 2015):

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 (Heider et al., 2012b). Consistent co-evolution of artefacts is a necessity adding complexity to an already non-trivial evolutionary process (Mens et al., 2005), occurring in both industrial (Hellebrand et al., 2014) and open-source contexts (Hunsen et al., 2015; Passos et al., 2015). Inconsistent modifications across the three spaces (variability, mapping, and implementation) may lead to the incapacity to derive products, code compilation errors, or dead code (Abal et al., 2014; Nadi and Holt, 2012; Tartler et al., 2011).

Recent studies (Neves et al., 2015; Passos et al., 2015) described typical changes occurring in such systems, giving insight on how each space could evolve, and revealing the relationship between the various artefacts. In (Passos and Czarnecki, 2014), Passos et al. proposed a dataset capturing the addition and removal of features.

Unfortunately, the most detailed change descriptions currently available (Neves et al., 2015; Passos et al., 2015) were obtained using extensive *manual* analysis of commits. Moreover, those studies focused on specific types of changes, such as addition and removal of features (Passos et al., 2015), or product line refinement scenarios (Neves et al., 2015). Consequently, the set of co-evolution scenarios documented is limited, and, saved

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

by performing a similar extensive manual analysis of a large number of commits, the identification of new scenarios remains difficult. Finally, the current state of the art offers neither data nor methods to obtain information on the prevalence of co-evolution in practice nor the frequency of those specific scenarios over a long period of time.

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

- (**S1**) 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 of the feature.
- (**S2**) A developer introducing a new feature to a subsystem is interested in finding how similar features were supported by similar subsystems in the past. Then, (s)he needs to look for changes in those subsystems, involving that such features.
- (**S3**) During bug triage, a maintainer is searching for a developer who might be able to resolve a specific issue. The maintainer would then be looking for developers with knowledge in the implementation on the possibly faulty features.
- (**S4**) 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 catalog presented by Passos et al. (Passos et al., 2015) and Neves et al. (Neves et al., 2015)
- (**S5**) Researchers working in the field bug prediction for highly configurable systems are interested in the relationship between variability changes and error-proneness. A database of detailed feature-related change information could facilitate their work.

Unfortunately, given the current state of the art, obtaining the necessary information require extensive manual analysis of changes and in-depth knowledge of the system under study.

We present in this paper the extension of FEVER (Feature EVolution ExtractoR) (Dintzner et al., 2016), 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.

We then manually compare the data obtained by FEVER and the commits as presented in the source control system to first evaluate the improvement in terms of change extraction accuracy obtained by the FEVER approach over its previous installment, and perform a second complete evaluation on a larger set of commits. We use this evaluation to answer the following research questions:

- RQ1: To what extent is the new version of FEVER more accurate in capturing feature-related changes?

- RQ2: To what extent does the new version FEVER data match changes performed by developers?

We use the resulting dataset to perform an exploratory study of feature evolution over 15 releases of the Linux kernel. We focus on the co-evolution of artefacts during feature evolution, in terms of affected spaces, under two different points of view: a feature perspective, focused on feature and the artefacts touched during their evolution, and an author-centric view, focused on commit authors and the spaces affected during maintenance operations. Using FEVER data, we aim at answering the following two research questions:

- RQ3: To what extent do artefact in different variability spaces co-evolve during the evolution of features?
- RQ4: To what extent are developers facing co-evolution over the course of a release?

While the tool we built to extract changes is centered on the Linux kernel, the approach itself is applicable to a larger set of systems (Berger et al., 2013a; Hunsen et al., 2015) 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.

Through this paper, 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 15 recent releases of the Linux kernel history (3.10 to 4.4 in separate databases), (4) an evaluation of the accuracy of our heuristics showing that we can extract accurately the information out of 87% of the commits, (5) we show that most (69.27%) of features evolve solely through their implementation, and that a majority of authors do not touch other spaces than the implementation space. Finally, the tool and datasets used for this study are available on our website.[2]

This study is an extension of our previous work on co-evolution of artefacts in highly variable systems (Dintzner et al., 2016). In this paper, compared to (Dintzner et al., 2016), we improved the model to better describe complex changes, with additional relationships between artefacts and information on artefact changes. We also improved the heuristics use to capture changes, leading to a higher change extraction accuracy. We also extracted a larger dataset, comprised of more detailed changes and over a longer period of time. Finally, research questions RQ3 and RQ4, on the quantitative aspect of co-evolution, are entirely new to this work.

We first provide background information on highly variable systems and the implementation of features in the Linux kernel in Section 2. Then, we present the FEVER approach, its change meta-model and the change extraction process in Section 3. We evaluate our approach by first comparing the performance of FEVER with its previous version presented in (Dintzner et al., 2016), and then provide a complete evaluation, including

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

new change attributes in Section 5. We show the usefulness of FEVER and the collected data in the aforementioned scenarios in Section 6. With the collected data, we perform an exploratory study of co-evolution occurrence in Section 7. We discuss our results and present the threats to the validity of our approach and complete study in Section 8. Finally, we present related work in Section 9 and conclude our work in Section 10.

2. BACKGROUND

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

2.1. 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 (Kang et al., 1990). In the context of the Linux kernel, the VM is expressed in the Kconfig language. An example of a feature in the Kconfig language is shown in Listing 4.1. 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, 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 4). The default value might be conditioned by an "if" statement.

Kconfig expresses feature dependencies using the "depends on" 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.

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

Listing 4.1: 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 presents 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.

2.2. 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 consider the following types of assets : implementation artefacts (i.e., source files), data artefacts (i.e., hardware description files), folders, and compilation flags. The addition of the mapping between a feature and code in a Makefile, as performed in the Linux kernel, is presented in Listing 4.2. In this example, the mapping is done between features and object files (but may link source code directly on occasion). We use the relationship between object files and source files to identify the mapped source file.

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 directly include "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, via conditional inclusion, or more complex mechanism relying on variables and file path reconstruction.

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

Listing 4.2: 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 conditions for artefacts.

2.3. ASSETS

Many types of assets exists, such as images, code, or documentation. We consider only configurable implementation assets (source files). We focus specifically on pre-processor based variability implementation (using #ifdef statements), which, despite known limitations(Spencer and Collyer, 1992), is still widely used today (Liebig et al., 2010). An example of an addition of a pre-processor statement is presented in Listing 4.3 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.

```
+ #ifndef CONFIG_SQUASHFS_FILE_DIRECT
 static inline void *squashfs_first_page
      (struct squashfs_page_actor *actor)
 {
  return actor->page[0];
 }
+ #else
+ static inline void *squashfs_next_page
+      (struct squashfs_page_actor *actor)
+ {
+ return actor->squashfs_next_page(actor);
+ }
+ #endif
```

Listing 4.3: 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 4.4.

```
#define DSL  CONFIG_DE2104X_DSL
```

Listing 4.4: Referencing to a value feature where the variable DSL will take the value associated with feature DE2104X_DSL

3. DESCRIBING CO-EVOLUTION

The objective of this work is to obtain a consolidated view of changes occurring to features and their implementation. This information is meant to be used for further analysis, and should capture the most relevant aspects of the changes regarding features and their evolution in the different spaces. In this section, we present the meta-model we use to describe feature-related changes in the different artefacts, and how we relate those changes to one-another. We illustrate the usage of the model with an 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". The commit[3] message for that change reads as follows:

```
iio: add APDS9300 ambilent light sensor driver

This patch adds IIO driver for APDS9300 ambient light sensor (ALS).
http://www.avagotech.com/docs/AV02-1077EN

The driver allows to read raw data from ADC registers or calculate
lux value. It also can handle threshold interrupt.
```

3.1. FEVER CHANGE META-MODEL

An overview of the FEVER change meta-model is shown in Figure 4.1. This overview highlights the different entities we use to describe what occurs in a commit, from a feature perspective.

[3] commit id: 03eff7b60d

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 code blocks related to features are modified, **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 FeatureEdit** 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. This relationship is established at a file level. The details of the changes within that artefacts are contained in the associated **Edit** entity.

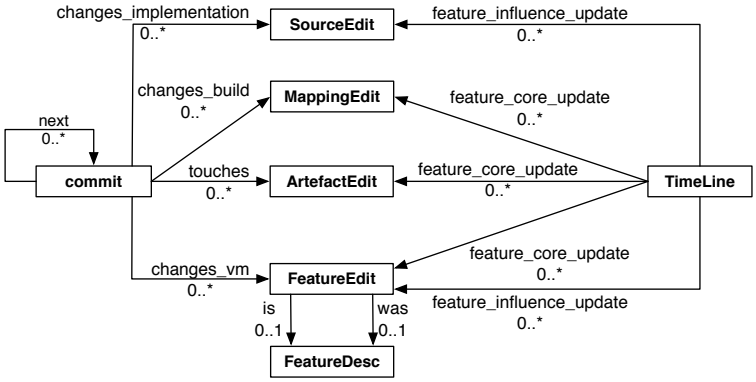

Figure 4.1: The FEVER change meta-model for feature-oriented change description

For a commit in the repository we record the commit id (sha1) to link 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 users-related information such as commiter and author of each commit. Table 4.1 summarizes the commit-related information stored in the FEVER database, examplified with the commit adding the "APDS9300" feature.

Attribute	Details	Example
hash	10 first digits of the commit unique ID	03eff7b60d
author	author's name	Oleksandr Kravchenko
commiter	commiter's name	Jonathan Cameron
message	complete commit message, including sign-offs	iio: add APDS9300 ambilent light sensor driver (...)
time	commit time	Sat Aug 03 19:40:37 CEST 2013

Table 4.1: FEVER **Commit** entity attributes

3.2. 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).

In our example, the developer added a new feature, APDS9300, to the variability model. The change that can be observed in the source control system is shown in Figure 4.2.

```
config ADJD_S311
        tristate "ADJD-S311-CR999 digital color sensor"
        select IIO_BUFFER
        select IIO_TRIGGERED_BUFFER
        depends on I2C
        help
         If you say yes here you get support for the Avago ADJD-S311-CR999
         digital color light sensor.

         This driver can also be built as a module.  If so, the module
         will be called adjd_s311.

+config APDS9300
+       tristate "APDS9300 ambient light sensor"
+       depends on I2C
+       help
+        Say Y here if you want to build a driver for the Avago APDS9300
+        ambient light sensor.
+
+       To compile this driver as a module, choose M here: the
+       module will be called apds9300.
+
 config HID_SENSOR_ALS
        depends on HID_SENSOR_HUB
        select IIO_BUFFER
```

Figure 4.2: Variability model change: addition of the feature APDS9300

The information recorded by FEVER on **FeatureEdit** entities are summarized in Table 4.2.

Attribute	Details	Example
name	name of the touched feature	APDS9300
change	change operation affecting the feature	ADDED
visibility	feature visibility to user during configuration	visible
type	type of the feature, defines its possible values	TRISTATE

Table 4.2: FEVER **FeatureEdit** entity attributes

The possible values for the "change" attribute are: "ADDED", "REMOVED", or "MODIFIED". The type attribute matches the configuration option type in the Kconfig language ("BOOLEAN","TRISTATE", "INT", "HEX", or "STRING"). The feature is either "visible" or "internal". Note that the type, and visibility information stored on the **FeatureEdit** entity correspond to the state of the feature after the edition takes place. For additional

information on the state of the feature before and after the change, one can refer to the **FeatureDesc** entities connected to the **FeatureEdit** entity.

The **FeatureDesc** entity captures the information presented in Table 4.3.

Attribute	Details	Example
name	name of the touched feature	APDS9300
type	feature type	TRISTATE
visibility	feature visibilty to the user during configuration	visible
depends on	dependencies of the feature	I2C
selects	the selected features	(none)
default values	default values, with conditions if any	(none)

Table 4.3: FEVER **FeatureDesc** entity attributes

For any feature change occurring at a variability model level, the change will be represented by a "FeatureEdit" entity, and at least one "FeatureDesc" entity in case of addition or removal, and at most two in the case of the modification of an existing feature.

3.3. MAPPING CHANGES

Regarding the evolution of the mapping, we are mainly interested in the evolution of the mapping between feature and asset. For this study, we consider the following types of assets: implementation artefacts, data artefacts, folders, and compilation flags. 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 to any type of assets, and the change affecting the target within that mapping (*added* or *removed*). Finally, if the asset is an artefact (file), then the change meta-model also includes the change to the artefact itself. 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). If the asset is not an artefact (such as a folder or a compilation flag) the value of the "artefact change" attribute is set to "NA".

In our example, the developer adds a mapping between the newly created feature and a newly added file by modifying an existing Makefile as shown in Figure 4.3. The information contained within the **MappingEdit** entity to represent this change are presented in Table 4.4.

```
#
# Makefile for IIO Light sensors
#

# When adding new entries keep the list in alphabetical order
obj-$(CONFIG_ADJD_S311)         += adjd_s311.o
+obj-$(CONFIG_APDS9300)         += apds9300.o
obj-$(CONFIG_HID_SENSOR_ALS)    += hid-sensor-als.o
obj-$(CONFIG_SENSORS_LM3533)    += lm3533-als.o
```

Figure 4.3: Mapping change: introduction of a new association between feature and asset

Attribute	Details	Example
type	element mapped to the asset (variable or feature)	FEATURE
feature	name of the feature involved	APDS9300
target	target of the mapping	apds9300.o
target type	type of the target (folder, flag, data, compilation unit)	COMPILATION_UNIT
mapping change	change to the mapping of the feature	ADDED
target change	change to the target entity within the feature's mapping	ADDED
artefact change	change to the artefact pointed to by the target	ADDED

Table 4.4: FEVER **MappingEdit** entity attributes

3.4. SOURCE CODE CHANGES

Feature related changes within source code, such as modifications to conditionally compiled blocks and feature references, are captured as **SourceEdit** entities. Features 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 code blocks are identified by the conditions under which they will be included in the final product. A change to such a block is represented by a **SourceEdit** containing the 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. An example of the code change is depicted in Figure 4.4.

```
+
+      return 0;
+}
+
+#ifdef CONFIG_PM_SLEEP
+static int apds9300_suspend(struct device *dev)
+{
...
+
+      return ret;
+}
+
+#define APDS9300_PM_OPS (&apds9300_pm_ops)
+#else
+#define APDS9300_PM_OPS NULL
+#endif
+
```

Figure 4.4: Source change: addition of conditionally compiled code blocks

In our example, two code blocks are added. Table 4.5 presents the information we obtain for the creation of the *else* fragment of this change. A similar entity is created for the first part of that new code block, the only different being the value of "interaction" attribute which would reflect the condition of the first block, namely "*defined(CONFIG_PM)*"

Attribute	Details	Example
Change	change to the code block itself, or the feature reference	ADDED
Interaction	presence condition of the block, or feature name for feature reference	!(defined(CONFIG_PM_SLEEP))
Code Edit	transformation of the code inside the changed block, "null" for references	ADDED

Table 4.5: FEVER **SourceEdit** entity attributes

3.5. TIMELINES: AGGREGATING FEATURE CHANGES

Changes pertaining to the same features are then aggregated into **TimeLine** entities. A **TimeLine** entity aggregates all changes pertaining to a single feature is a number of commits - this includes modification of artefacts mapped to the feature in question, FeatureEdit, MappingEdit or changes to conditionally compiled code blocks whose conditions refer to that feature. 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 changes*.

A *feature core update* 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 update* indicates that the feature is playing a role in the behaviour of another feature. This occurs in two contexts: in the source code, as part of a **SourceEdit**, or in the variability model as part of a **FeatureEdit**.

Figure 4.5 depicts all entities and relationships used to describe the changes occurring in single commit 03eff7b60d. This is a partial view of the complete database. When fully expanded, the "PM_SLEEP" **TimeLine** points to any **Edit** entity which describe changes to the "PM_SLEEP" feature across an entire release. By navigating through those relationships, one can easily find what transformation occured on each feature and retrieve contextual information regarding this change.

In Figure 4.5, three **TimeLine** entities are depicted in pink, on the right hand side of the diagram, annotated with the feature name. The first one relates to the feature that was introduced. We can see that the "APDS9300" node is connected to the **FeatureEdit**, in red in the diagram marked with the feature name "APDS9300", the **MappingEdit** in gray annotated with the name of the changed target (apds9300.o), and an **ArtefactEdit** (represented by a small gray dot for visibility purpose) 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.

We can also 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. Finally, a TimeLine for feature I2C point to the FeatureEdit introducing feature APDS9300. Note that, APDS9300 depends on I2C, and that relationship is new. For that reason, in this commit the influence of feature I2C was changed, and does not affect APDS9300.

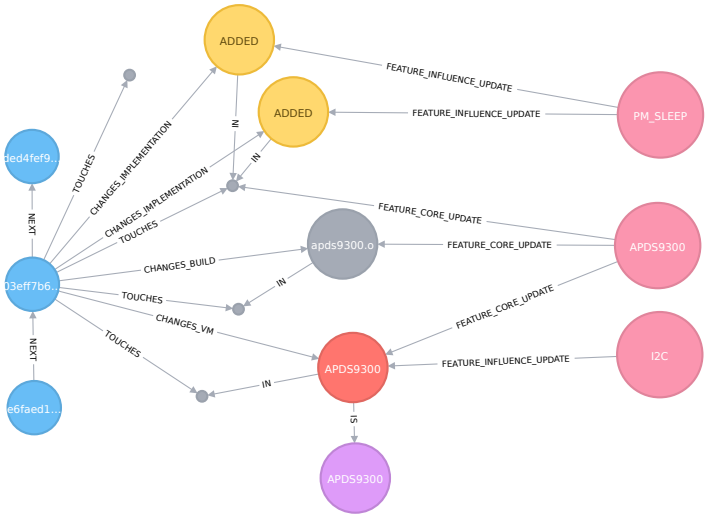

Figure 4.5: FEVER representation of commit 03eff7b60d - all entities and relationships. For readability purposes, **ArtefactEdits** are represented by small unlabelled gray dots. From top to bottom, they represent edits to the following files: a documentation file, the source file containing the behavior of feature APDS9300, the Makefile containing the new mapping, and the Kconfig file containing the new feature declaration.

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.

4. POPULATING FEVER

4.1. 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 4.6 presents an overview of the change extraction 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 using the mapping presented in Table 4.6. Compared to our previous work (Dintzner et al., 2016), we adjusted our artefact identification heuristics regarding source files, with a more restrictive expression on ".S" files (rather than ".S*"). We also include binary files (libraries), which were previously not taken into account.

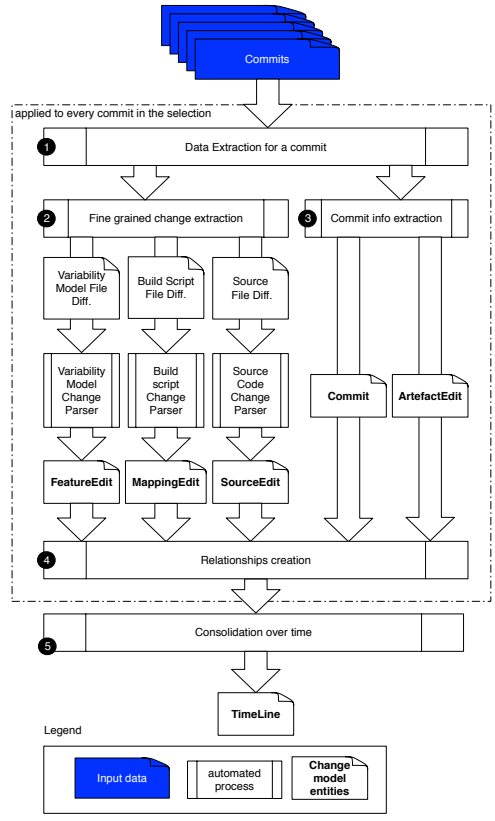

Figure 4.6: Overview of the FEVER change extraction and consolidation process

Step 2 performs the artefact-specific data extraction processes. The next subsections (Section 4.2,Section 4.3, and Section 4.4) 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[4] 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. We provide an evaluation of the accuracy of those heuristics in Section 5.

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 (i.e., documentation, or scripts), represented by **ArtefactEdit** entities.

[4] http://wiki.eclipse.org/EMF_Compare

Artefact type	Expression used for identification
V.M. file	"Kconfig.*"
Build file	"Makefile.*","Kbuild.*","Platform.*"
Source file	"*.c", "*.h", "*.s", "*.S"
Binary file	"*.dll","*.so","*.a","*.lib"
Data file	"*.dts","*.dtb"

Table 4.6: Artefact types: regular expression used to identify the different types of artefacts

In *Step 4*, FEVER creates 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 to keep track of the sequence of changes, and feature **TimeLine** entities with their respective relationships to edit entities. This is done by navigating 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.

We continue this section by describing the heuristics we used to extract feature related changes. Those heuristics are based on multiple sources of information, namely the work of Neves et al. (Neves et al., 2015), the work of Passos et al. (Passos et al., 2015), the Linux official documentation, and finally the authors' expertise (Dintzner et al., 2015a; Passos et al., 2015).

4.2. EXTRACTING VARIABILITY MODEL CHANGES

We describe in this section the artefact-specific change extraction process (Step 2 in Figure 4.6) that takes place when a commit contains changes to the variability model of the system.

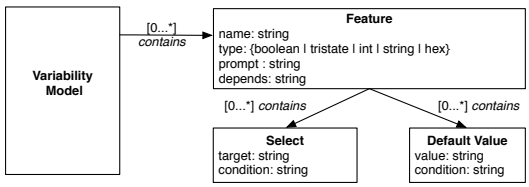

Figure 4.7: Representation of the variability model used for change extraction

The characteristics of the changed features that we focus on are their type (Boolean or value-based) and the change affecting the feature. We first reconstruct two instances of the VM depicted in Figure 4.7 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 with respect to the complete Linux variability model. The extraction process follows the FMDiff approach (Dintzner et al., 2015a), 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 (Dintzner et al., 2013). We use this XML representation of the Linux VM to build the model shown in Figure 4.7.

We then use EMF Compare to extract the differences and compile the information in a **FeatureEdit** entity. To successfully compare two model instances, FEVER needs to provide EMF with the capability to determine that two features in the two model instances are the same entity. For this, we rely on the feature name as a unique identifier during the model comparison phase.

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. As mentioned, the “source” statement in the Kconfig language is used to link Kconfig files together. Such statements can be used in combination with other constructs, such as menus, or “if” blocks. In this situation, the presence condition of the menu, or the condition of the “if” blocks, in practice applies to all features within “sourced” file, and any of the files it might “source” itself. By working on a file level (touched Kconfig file), FEVER will not capture such complex changes.

With respect to our previous work, we now handle cases where two features within the same file have the same name. Whereas the previous heuristic yielded a number of false positive, such cases are now handled by suffixing feature names by an index if a feature name is encountered twice (or more) when rebuilding the EMF model we use for change extraction.

4.3. EXTRACTING MAPPING CHANGES

We describe in this section the artefact-specific change extraction process (Step 2 in Figure 4.6) that takes place when a commit contains changes to the mapping between features and assets.

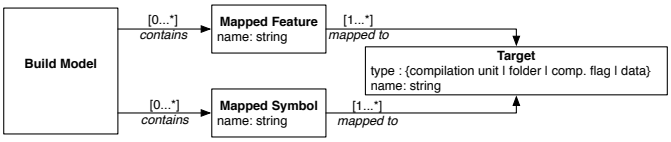

Figure 4.8: Representation of the feature-asset mapping used for change extraction

Similar 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 4.8.

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",".c" or ".h"; mapped data artefacts in the Linux kernel are identified by the extensions ".dts", ".dtb"; compilation flags either start by the follwing strings "-D", "-L", "-m", or "-W", "-I", "-f". We identify folder names by "/", or single words, not containing any special characters nor spaces.

Makefiles may contain lists of assets that will be included in the compilation as soon as the Makefile itself is included. Those assets are assigned to Makefile variables whose names depend on the implementation of the build process. In the Linux kernel, those are identified by [5]: "obj-y","lib-y","ccflags-y","asflags-y", and "ldflags-y". When we find assets associated with such variables, we map them to a temporary variable, using the following convention: we use the key word "guarded_" and append the name of folder containing the Makefile. We later use this naming convention with the extracted information on features mapped to folders to assign the changes of such Makefile variables to the appropriate feature(s).

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 4.5, 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 4.5, 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 5.

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

Listing 4.5: Example of an "ifeq" statement and aliases used in Makefiles

Finally, FEVER uses a Linux specific heuristic for mapping files contained within specific folders. Part of the mapping between feature and folder is done using variable names, and dynamic path reconstruction. In general, FEVER does not attempt to recover this mapping, but for a specific set of folder in the Linux kernel, namely the architecture folders, this mapping is important. Upon compilation, the chosen hardware architecture of the kernel forces the selection of a given subfolder of the "./arch" folder. There is no explicit declarations of that mapping in any Makefile (it uses variables and name reconstruction). For this reason, FEVER assumes that any file within the "arch/x86" folder maps to feature "X86" if no other mapping is found. The accuracy of this heuristic to recover the link between features and artefacts is evaluated in the next section as the *feature-file mapping* change attribute.

Our model reconstruction is based on heuristics and therefor do not take into account all the possible constructs used in the Linux kernel to link artefacts to features, however, FEVER focuses on those mentioned above. The constructs that FEVER does not

[5] https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt

capture are based on variable name manipulation, to build artefacts names (e.g. folder names, or file names), or combining lists of artefacts together. Then, as mentioned in Section 2, 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.

Once the two instances of the model are reconstructed, we use EMF Compare to extract the differences between them, giving us the list of feature mappings that were added or removed in that commit. For the comparison of two instances of our mapping model, we use the name of features as unique identifiers.

From the earlier version of this work, we now capture mapping between features and more artefacts, and our coverage of compilation flags is more comprehensive.

4.4. EXTRACTING IMPLEMENTATION CHANGES

We describe in this section the artefact-specific change extraction process (Step 2 in Figure 4.6) that takes place when a commit contains changes to the implementation (source code).

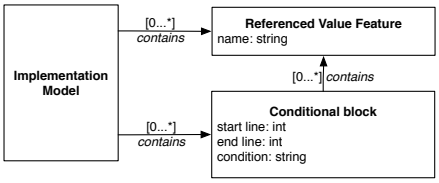

Figure 4.9: Representation of the feature-asset mapping used for change extraction

At the implementation level, we consider changes to #ifdef blocks and changes to feature references in the code, as presented in Section 2. To extract those changes, we rebuild a model of each implementation file in its before and after state following the model presented in Figure 4.9.

To rebuild the models, we rely on CPPSTATS (Liebig et al., 2010) 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. Note that we report reference changes once per feature and per file.

We then use EMF Compare to compare the two models and build the **SourceEdit** entities. For this comparison, FEVER needs to use a unique identifier for each code block contained within a source file. The condition on a block may not be unique, and hence cannot be used to uniquely identify a block in two versions of the source model. The location of the block within the file may change during a commit without the block being changed itself (i.e., if code is added or removed above it). FEVER uses a combination of the condition of the block combined with its content (the actual code) as a unique

identifier. This proved to be an efficient technique, but in the context of the Linux kernel a number of files contain identical code blocks, with the same block condition. While this may seem surprising, one may consider a logging mechanism: if the logger feature is selected, write an entry in the log file. This might be repeated in multiple functions in a file. As a result, the EMF comparison process cannot correctly identify changed blocks and returns a number of false positive changes. To compensate for this, we add indices to the identifier of code blocks when we find such duplication.

We determine the code changes occurring inside #ifdef blocks to compute the value of the "code edit" attribute of **SourceEdit** entities. This is performed as a separate step, once we found the changed code blocks. 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.

FEVER extracts and records changes to all conditionally compiled code blocks - whether features play a role in their presence condition or not. Changes to code blocks that are not tied to any feature will be captured as **SourceEdit**, but such entities will not be linked to any **TimeLine** in the next step of our process.

By comparison with our previous work, we enhance the source change extraction process by taking into account cases where code artefacts contain identical code blocks, containing identical code. Such situations caused errors during the EMF comparison process and are dealt with as explained in this section.

4.5. CHANGE CONSOLIDATION AND TIMELINES

The final step consists 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 strategy:

- if a feature is touched in the VM, mapping or source file, the corresponding **Edit** entity is associated with a **TimeLine** with a "core update" relationship.
- if a feature A is added from another feature B's attribute (as part of a constraint), then the **FeatureEdit** entity representing this change is connected to the feature **TimeLine** with an "influence update" relationship if feature A did not participate at all in the definition of B before the change.
- if a feature A is removed from another feature B's attribute (as part of a constraint), then the **FeatureEdit** entity representing this change is connected to the feature **TimeLine** with an "influence update" relationship if feature A no longer participate at all in the definition of B after the change.
- if a feature is part of the condition in a **SourceEdit** entity, the **SourceEdit** is connected to one **TimeLine** entity per feature present in the condition with an "influence update" relationship;
- if an artefact is touched, it is linked to the **TimeLine** entity of the feature to which it is mapped with a "core update" relationship. This is done for each feature mapped to the file.

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. To support systems which do not follow Linux naming convention (the CONFIG_ prefix used in Makefile and the source code), we also extract the list of features present at the beginning of the studied time-frame. For both the initial feature list and initial mapping, we rely on the FEVER parser to obtain the information by invoking it for every Kconfig file and Makefile present in the system.

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 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.

5. 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.

The objective is two-fold. First, we aim at evaluating how the changes to the heuristics impacted the accuracy of the FEVER approach. Secondly, we aim at providing a complete evaluation of the FEVER approach and its accuracy, including all new change attributes, against a larger and more representative set of commits as before.

Throughout this section, we consider that a FEVER change description is "accurate" if the changes performed by developers are captured correctly by FEVER as described in the previous section. We evaluate the accuracy of the approach in terms of precision and recall with respect to changes performed by developers on the observed artefacts.

With this work, we improved on the existing FEVER prototype (Dintzner et al., 2016) in several ways. Section 3 described the FEVER approach with its improvements. From the initial version of this work, we improved the following aspects of the approach:

- heuristics for code reference identification
- heuristics for code changes within modified code blocks
- heuristics for asset-feature mapping identification (compilation flag, default list, and artefact extensions management)
- the build change model to support more types of artefacts (namely data artefacts)

- the build change extraction to include artefact changes when describing mapping changes
- the timeline model to include "influence updates" on feature changes

With those changes, FEVER captures more information than before, and should be able to capture previous information more accurately. This leads us to formulate the first research question driving this evaluation:

> RQ1: To what extent is the new version of FEVER more accurate in capturing feature-related changes?

However, the enhancements of FEVER also include the addition of new information regarding feature-related changes. The overall accuracy of the tool, should also be evaluated. We propose to answer the following research question:

> RQ2: To what extent does the improved FEVER data match changes performed by developers?

To assess whether the FEVER data matches the content of commits, we perform here a two-steps evaluation. First, we apply FEVER on the commits used in (Dintzner et al., 2016) and compare the results obtained during the first evaluation of FEVER and the improved algorithm. Then, we perform a second, entirely new evaluation on two different releases using a different heuristic to select commits.

For both steps, the evaluation is performed manually and consists in comparing the content of the FEVER database with changes performed by developers. We first present how this comparison is performed. Then, we present the results of the replication of the evaluation and finally present the results of the evaluation on the new set of commits.

5.1. 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 compared 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 checked that the different **Edit** entities and their attributes can be explained by the changes observed in Git. Conversely, we ensured that feature-related changes seen in Git have a FEVER representation. At variability model level, we checked 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 checked that the pairing of features and files is accurate and that the type of targeted artefact is also correct. Special consideration was 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 takes into account a number of such constructs as mentioned in Section 3, but not all possible ones. In cases where a mapping change can be observed in a Makefile, but FEVER does not report any change, we checked in the Makefile hierarchy if a feature should have been mapped to that change. If, during the manual inspection, we reached the root folder of the Linux file hierarchy and we have not encountered any explicit declaration of a link between the changed mapping and any feature, we considered that this change could not have been mapped by FEVER, and FEVER should not report any feature-related mapping change. For instance, a developer modifies "./mm/Makefile" (memory management), and adds a compilation unit to the "obj-y" variable. We see that the file "./mm/Makefile" is not constrained by any feature in the root "./Makefile" of the kernel source tree. Hence, we consider that FEVER cannot map this mapping change to any feature, and should not report it. We emphasize that FEVER will still report that the Makefile has been touched in the form of a **ArtefactEdit**, but no **MappingEdit** entity should be present.

At the code level, we checked that the blocks seen as touched are indeed touched, and we compared the condition of each block. Then, by inspecting the patch, we validated that the code changes within the blocks were correct.

Regarding **TimeLine** entities, we did not check whether all relevant changes in all commits were indeed gathered into **TimeLine** entities. We made the assumption that if **TimeLine** entities were properly linked in the commits we checked, then the algorithm is correct, and the check on the complete release is therefor unnecessary. We also kept track of the commits for which all extracted information is accurate, giving us an overview of the accuracy on a commit basis.

5.2. REPLICATION

In our previous work (Dintzner et al., 2016), we evaluated our tool as follows. 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 (out of 11,907 and 13,288 respectively). The selection of commits in those two releases was performed as follows: we randomly selected 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 all three spaces. During the evaluation, we ignored merge and release tag commits.

To evaluate our improved algorithm, we performed the same analysis over the same set of commits using the enhanced FEVER prototype and compared the results obtained with what was previously established. Table 4.7 presents a comparison between the previous precision and recall obtained on change attributes as well as the precision and recall for the new algorithm.

In addition to the information presented in the table, our evaluation showed that the percentage of commits for which FEVER correctly extracted all change attributes increased from 82,7% to 85,3%.

Let us first discuss the differences in terms of sample change between the two evaluations. We note that between the two evaluations, few sample size are exactly the same.

	Reference algorithm (Dintzner et al., 2016)			Current algorithm		
Attribute	Sample	Precision (%)	Recall (%)	Sample	Precision (%)	Recall (%)
VM operations						
change: *added*	208	100	100	206	100	99
change: *removed*	73	100	100	74	100	100
change: *modified*	140	80	100	138	81.4	98.6
Mapping operations						
target: *folder*	17	100	94	17	100	100
target: *compilation unit*	437	100	98	430	100	99.8
target: *compilation flag*	10	67	60	14	100	100
mapping change: *added*	278	99	97	271	98,9	98.9
mapping change: *removed*	84	100	95	133	100	100
mapping change: *modified*	98	100	98	68	98.6	100
target change: *added*	326	99	97	328	98.2	97.9
target change: *removed*	133	100	97	139	100	100
file-feature mapping	622	81	97	728	93.4	92.6
Source operations						
block change: *added*	381	81	97	321	98.7	92.6
block change: *removed*	229	100	99	230	100	97.8
block change: *modified*	237	97	99	233	96.3	100
code change: *added*	365	99	97	307	99,0	98.4
code change: *removed*	195	99	99	190	96.4	98.4
code change: *edited*	237	96	99	236	95.9	100
code change: *preserved*	46	32	83	45	93.2	91.1
reference change: *added*	6	100	83	106	100	100
reference change: *removed*	7	88	100	5	83.0	100
TimeLine	743	93	98	11225	95.5	97.5

Table 4.7: Comparison of accuracy of the initial FEVER heuristics (Dintzner et al., 2016) with its new version.

For instance, the first evaluation recorded 208 added features, but the second one found a total of 206. The evaluation process being inherently manual, it is reasonable to observe slight differences (as in the variability model changes for instances). However, variation of sample size is more significant for the following attributes: feature-file mapping, block changes added, added code, added references, and timelines. Regarding the feature-file mapping, the new version of FEVER attempts to resolve the mapping of more files - rather than focusing only on source code. Previously, FEVER did not do so for files located within an "include" folder (at any level of its path). Changes to files in folder such as "arch/.../include" are now mapped.

The variation in terms of "block changes: added blocks" and "code change:added code" are related. During the first evaluation, we found 381 added code blocks (block changes: added blocks) with 365 occurrences of new code blocks containing only new code (code change:added code), while during the second evaluation the number of added code blocks dropped to 321, and the number of code blocks with added code dropped to 307. The difference between the two values stems from changes obtained from a single commit. A file with the extension ".S_shipped" containing a large number (60+) of added interactions was included in the initial evaluation. We adjusted the algorithm to identify files, enforcing strict file extension (.S), hence the file was ignored during the second evaluation. This results in less added code blocks, and the less added code blocks containing only new code. While this raises the question of which artefacts one should consider during the experiment, it does not undermine the ability of FEVER to capture

accurately code changes from within a well defined set of artefacts.

The number of added references increased significantly between the two evaluations. Once again, explanation for this difference is contained within a single commit[6] where a hundred features are added, and then referenced in the code. During the first evaluation, those references where incorrectly identified as local macros by the tool and the reviewer, and not noted as added references. During the second review, with the updated algorithm, the references were correctly identified by FEVER as feature references. A deeper analysis of the code and the related artefacts showed that those were indeed feature references and should be recorded as such.

Finally, with the improved approach,**TimeLines** now may be created as the result of a feature relationship change. Since this was not taken into account during the first evaluation, the number of **TimeLines** obtained with the improved algorithm (11,225) is *de facto* larger than during the first evaluation (743).

Despite those differences, the results in Table 4.7 indicate improvement of the accuracy of most change attributes related to mapping and code changes. The most significant being the detection of preserved code inside changed code blocks (from a precision of 32% to 93,2%) and the detection of changes to compilation flags during mapping evolution (from a precision and recall of 67% and 60% to 100%). Code change capture was improved by avoiding false positives when multiple code blocks were identical. The detection of compilation flag changes was improved by capturing changes to compilation flags not mapped "directly" to a feature, but indirectly (the flag is mapped to an internal variable and will be activated when a guard feature is selected).

With this information we can answer our first research question, RQ1: To what extent is the new version of FEVER better at capturing feature-related changes?

> The overall accuracy of FEVER slightly improved (by 2,6%), while the ability to capture certain change attributes increased significantly (by more than 30%).
>
> The changes to the heuristics used by FEVER lead to an improvement over its previous version.

While this increases our confidence in FEVER's ability to capture changes, the improved algorithm allowed us to capture change in artefacts and feature relationships that were not taken into account before - hence, not covered in this comparison. Moreover, we used for this comparison the same set of randomly selected commits as in our previous work (Dintzner et al., 2016). However, the methodology used to build this set did not allow for commits not affecting any feature to be included in the evaluation, which, in our opinion creates a bias in the evaluation. We continue the evaluation of FEVER by performing a complete evaluation, including new attributes on a more complete and different set of randomly selected commits.

5.3. EVALUATION ON A NEW SET OF COMMITS

The results of the previous sub-section highlight improvements on the ability of FEVER to capture certain types of changes. However, we extended the change model to capture

[6] 206f060c21

additional change information, as presented in the beginning of this section.

To evaluate the improved FEVER algorithm, we extended the evaluation of the data used in the replication presented above (300 commits) to cover the additional changes and created a new dataset from two releases using a different random selection approach (510 additional commits). For the additional dataset, instead of three groups of commits affecting different spaces, we randomly selected commits from five different groups: 51 commits not affecting any artefact, 51 commits affecting arbitrary artefacts, 51 commits affecting at least the variability model, 51 commits affecting at least the mapping, and finally 51 commits affecting at least code blocks, for a total of 255 commits per release. With this approach, we ensure that every commit within the FEVER database may be selected. Consequently, the complete dataset used for this evaluation is comprised of 810 commits, from 4 different releases (150 commits from release 3.12, 150 commits from release 3.13, 255 commits from release 3.14, and finally 255 commits from release 4.2).

FEVER does not capture changes inside merges. The rationale behind this decision is to avoid capturing changes multiple times: once when they are implemented by their original authors, and possibly a second time if the merge operation results in a conflict (same file modified twice). During our evaluation, we checked whether some information was missed by skipping merge commits altogether. We used the following methodology: we inspected a subset of the merge commits and checked that all changes that occurred can be found within the parent commits - i.e. all modifications pre-existed, they are simply integrated together. We identify “new content” in merge commits by using the following “git log” command to visualize the changes:

git log <commit_hash> -p –cc

The “-p” option displays the patch, and "-cc" displays the patch “diff” from all parents simultaneously. Using this view of the patch, we searched for content added or removed from all parents. Practically, this amounts of searching for lines in the “diff” where the number of “+” or “-” symbols at the beginning of modified lines of text equals the number of parents.[7] Given that FEVER omits merge commits, any of such change is accounted for as a false negative for the relevant change attribute during the evaluation. Table 4.8 summarizes the results for the 4 datasets, comprised of a total of 810 commits.

The results show that, for a majority of attributes (26 out of 27), FEVER precision and recall is at least of 88%. On the other hand, we note that detection of reference changes can be problematic. During this evaluation, we found two cases where developers created local variables (using the #define C directive) whose name matched feature naming convention (CONFIG_ prefix). This explains the lower precision, but FEVER still exhibit for this change a high recall of a 100% when capturing removals of feature references.

With this information we can now answer our second research question, RQ2: To what extent does the new version FEVER data match changes performed by developers?

> The results showed that the data collected by the new version of FEVER matches the changes performed by developers in 87% or more of the commits. The newly in-

[7] https://git-scm.com/docs/git-log

cluded change attributes (artefact changes, “data” artefact types) are captured with a high accuracy (of at least 80%).

Those results give us confidence on the viability of the FEVER approach, and in the quality of the extracted data. We proceed to explore usages of the dataset, before continuing with an exploratory study of co-evolution of artefacts in the context of feature evolution in Section 7.

Attribute	Population	Precision (%)	Recall (%)
VM operations			
change: *added*	309	100	98.7
change: *removed*	88	100	98.9
change: *modified*	293	89.8	98.6
Mapping operations			
target: *folder*	52	100	98.1
target: *compilation unit*	735	99.3	95.6
target: *compilation flag*	32	100	100
target: *data*	61	100	100
mapping change: *added*	506	98.4	95.1
mapping change: *removed*	201	100	90.0
mapping change: *modified*	180	97.7	92.8
target change: *added*	644	98.7	94.6
target change: *removed*	224	99,1	100
artefact change: *added*	366	98.2	91.5
artefact change: *removed*	113	99.0	89.4
artefact change: *modified*	31	100	80.6
artefact change: *untouched*	290	88.7	92.4
artefact change: *NA*	82	100	98.8
file-feature mapping	1650	95.1	93.5
Source operations			
block change: *added*	656	99.4	93.5
block change: *removed*	355	100	97.2
block change: *modified*	529	95.6	99.6
code change: *added*	583	99.1	97.9
code change: *removed*	271	97.0	96.7
code change: *edited*	556	95.3	99.3
code change: *preserved*	124	95.7	88.7
reference change: *added*	117	99.2	100
reference change: *removed*	9	69.7	100
TimeLine	2367	97.1	97.5
Correct commits	810	87.2%	

Table 4.8: FEVER change extraction accuracy evaluated on 810 commits

6. FEVER USAGE SCENARIOS

In this section, we illustrate how using FEVER or the data collected using the approach can be of use to developers, maintainers and researchers in the scenarios **S1** to **S4** mentioned in the Section 1.

The FEVER data is stored in a Neo4j graph database.[8] 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 section are written in the Cypher query language.[9] It is understood that, in a practical situation, an integration with development tools would be more suitable than relying on direct Cypher queries.

6.1. FEVER FOR SOFTWARE DEVELOPMENT ACTIVITIES

In scenario **S1**, we consider the work 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 [10] 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:

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

This query returns two commits. The first commit (id:69f055) mentioned in the release note is associated with a **FeatureEdit** entity denoting the addition of a feature. The second commit (id: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 managers were, in fact, incorrect. We argue that a dataset such as FEVER would provide release manager with more accurate information on changes that were performed by developers and may have prevented this erroneous entry in the release notes.

In scenario **S2**, a developer is about to introduce a new driver for a touch-screen supporting the power management "SLEEP" feature. The developer might want to know how such support was implemented in other drivers and compare it with its own implementation. Using FEVER, he queries the database for commits where a new feature (f1) is added (fe.change ="ADDED"), and 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)
```

[8] http://neo4j.com/
[9] http://neo4j.com/docs/stable/cypher-query-lang.html
[10] http://kernelnewbies.org/Linux_3.13

```
where f2.name = ''PM_SLEEP''  and fe.change = ''ADDED''
return f1,f2, distinct c;
```

When ran against database containing commits of release 3.14 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" and might consider using this as an example to drive his own development.

FEVER can also be of use in our third scenario **S3** in the context of bug triaging. Let us consider the bug #928561 reporting issues with keyboards mentioning that "*multimedia and macro keys are not working*".[11] The bug report author provide traces and logs pointing to issues with the Linux Human Interface Devices (HID) subsystem. This issue was fixed by and the patch was introduced in the kernel in release 3.12. In the FEVER database for release 3.12, we run the following query to see who among the commit authors committed the most changes affecting HID related features.


```
match (c:Commit)-->()<--(t:TimeLine)
where t.name=~"(?ism).*HID.*"
return distinct (c.author), count(c)
order by count(c) desc;
```

The name of the developer who analyzed and fixed the issue comes first in the results, with 22 commits affecting "*HID*" features - among which one corresponds to the patch fixing the keyboard issue. In second place, we find an official maintainer for three of kernel subsystems with 17 commits, followed by another official maintainer for two HID related subsystems and the name of a Linux branch manager with 16 commits each affecting such features. It is interesting to note that the names of the developers who fixed the issue in question are not present in the official maintainers list of the kernel for releases 3.11 nor 3.12. Through this scenario, we suggest that the FEVER database can be of use to identify feature expertise, and possibly facilitate bug triaging (Matter et al., 2009). A maintainer in charge of bug triage may use a simple query with information on potentially faulty features to find which developers can provide insight on an issue or even fix it.

6.2. FEVER FOR SOFTWARE ENGINEERING RESEARCH

In scenario **S4**, a researcher in the domain of evolution of highly variable software systems is interested in the typical structure of feature related changes. For instance, he would like to observe the occurrences of the introduction of abstract features, in the sense of Thuem et al. (Thuem et al., 2009): a feature only exists in the VM. Using FEVER, 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)
```

[11] https://bugzilla.redhat.com/show_bug.cgi?id=928561

```
 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. The addition of an abstract feature has not yet been described as a co-evolution pattern, and further analysis is necessary to fully describe such changes. Nonetheless, this illustrates how FEVER can be of use to discover patterns or identify instances of known patterns. An earlier version of FEVER was used by Sampaio et al. to facilitate the identification of instance of changes affecting certain spaces in the context of their work on safe evolution templates (Sampaio et al., 2016).

In scenario **S5**, we consider the work of a researcher focusing on variability related bugs (Abal et al., 2014) and bug prediction (Giger et al., 2011). The data captured by FEVER may reveal information on features involved in bug-fixing commits. A basic approach would consist in using regular expression on commit messages to identify bug-fixing commits. Using this, one can identify features involved in bug-fixing commits using the following query:

```
match (c:commit)-->()<--(t:TimeLine)
where not c.message =~ "(?ism).*copyright notices.*"
and c.message =~ "(?ism).* bug.*"
or c.message =~ "(?ism).* error.*"
or  c.message =~ "(?ism).* fix.*"
or c.message =~"(?ism).* revert.*"
return t.name, count(distinct c);
```

We note that Tian et al. devised a methodology to identify bug-fixing commits in the Linux kernel (Tian et al., 2012). Combining such an approach with FEVER should yield more accurate results than the query presented here. However, with such a simple query, one can identify which features are more error-prone than others. It would be interesting to see if the number of features involved in a commit influences the bug-proneness of commits.

Finally, the data provided by German et al. (German et al., 2015) 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.

7. CO-EVOLUTION IN LINUX

In this section we explore the data collected by FEVER over 15 releases of the Linux kernel. Given the relatively high accuracy of the approach established in Section 5, we can rely on FEVER data to explore co-evolution of artefacts in the context of feature evolution in the Linux kernel.

The state of the art on feature-oriented co-evolution of artefacts in highly configurable software systems focused on specific changes (Neves et al., 2015, 2011; Passos et al., 2015). Those studies were performed using manual analysis. While those provide

relevant and important knowledge on change scenarios, little information can be found on their occurrence in large systems. In this section, we report on an exploratory study of the feature-oriented co-evolution of artefacts in the Linux kernel.

We argue that quantitative information on the frequency of co-evolution over the evolution of a complex system would allow tool developers and researchers to determine how relevant the support of co-evolution for the evolution of such systems is. How often is co-evolution occurring, and how many authors actually face co-evolution during their development tasks? What percentage of the touched features actually evolve in multiple variability spaces? And when they do, which spaces are more frequently involved? Should a developer provide tool support for co-evolution, and what should be its main focus to help in a majority of cases? This leads us to formulate the following two research questions:

- RQ3: To what extent do artefact in different variability spaces co-evolve during the evolution of features?
- RQ4: To what extent are developers facing co-evolution over the course of a release?

With the first question, we can obtain an estimate of how likely co-evolution is from a technical perspective. If a feature evolves during a release, how likely is it that this evolution will imply the modification of multiple types of artefacts? With the second question, we aim at estimating the potential audience for tools and techniques targeting co-evolution issues. Provided a simple and efficient method can be devised to guarantee correct feature-oriented co-evolution of artefacts, what percentage of the development team would actually benefit from it?

To put our results into perspective, we first provide our readers with general information on the evolution of the Linux kernel as captured by FEVER over the studied period of time. The dataset collected with FEVER covers 15 releases of the Linux kernel, starting at v3.9 (April 2013 - first extracted commit) until v4.4.(January 2016 - last extracted commit). A release of the Linux kernel lasts for approximately six weeks.

Release	3.10	3.11	3.12	3.13	3.14	3.15	3.16
Number of commits	14737	11851	11906	13288	13415	14871	13830
Number of authors	1433	1304	1362	1400	1481	1535	1513
Number of features	12511	12603	12780	13022	13134	13297	13453
Number of timelines	5208	4397	4424	4581	4503	4960	4099

Release	3.17	3.18	3.19	4.0	4.1	4.2	4.3	4.4
Number of commits	13331	12361	13652	11306	12965	14750	13282	14082
Number of authors	1461	1507	1495	1495	1576	1630	1607	1636
Number of features	13602	13631	13802	13932	14427	14217	14458	14607
Number of timelines	4322	3797	5131	3432	4082	4316	4159	3967

Table 4.9: General information on the Linux kernel development: number of commits, authors, features, and FEVER **Timelines** over the studied period of time.

7.1. METHODOLOGY

Before proceeding, we first provide general information on the studied releases. Table 4.9 presents the number of features at the beginning of each release, the number of authors, the number of commits, and the number of **TimeLine** entities. The number of features at the beginning of the release is obtained by using the initial feature list produced for the extraction process. The number of **TimeLine** was obtained by querying the FEVER databases, representing the number of features that evolved during that release. The number of commits and authors were obtained by querying the FEVER database and cross-checked using "Git".

We then proceeded as follows. We built a number of queries to identify features, the spaces in which they evolve and the involved authors. We ran the queries on each extracted release of the Linux kernel and dumped the results in a series of .csv files. For each commit we extracted the type of artefacts affected by the commits as well as the authors. To identify authors, we used the author name, as reported in the Git repository - this information is stored as part of the commit entity in FEVER. We also consolidate the collected information over time. This allows us to contrast the evolution of feature and variability authorship in each release with the evolution of feature and variability space authorship over multiple releases (15 in this case). To do so we aggregate the collected information by feature (identified by their name), and authors (identified by their name as well). By doing so, we avoid biases caused by complex co-evolution over time. For instance, a feature is touched in the code in six releases, but its mapping or variability model representation change in seventh. Over time, this should be considered as a change to all spaces, where on a release level, we would record a changes in the source code only, or V.M. and build only - which would be correct but partial. We then imported this information into a spreadsheet editor to compile the results.[12]

As noted in previous work on mining social information from software repositories (Bird et al., 2008; Kouters et al., 2012), authors are likely to use aliases and submit commits using different email addresses. In this work, we relied on the author's name, as stored in the Git repository and did not take aliases into account. We evaluated the possible bias caused by aliases on our study by performing a manual analysis of author's name in release 4.4. To identify aliases, we searched among the list of author names duplicated names and first name. We then decided whether two names are likely to point to the same person using the following strategy: for each name we took into account the following variations mentioned by Kouters et al. (Kouters et al., 2012): *odering, diacritics, nicknames, middle initials and middle name*, and finally *irrelevant incorporation, emails instead of name*. This analysis of author's name in release 4.4 revealed that, out of the 1636 authors, 53 recorded author names are aliases, accounting for 3.23% of author names.

7.2. RESULTS: FEATURE CO-EVOLUTION OVER TIME

The results of our quantitative analysis of co-evolution of features in the Linux kernel are presented in Table 4.10. This table summarizes, for each release, the space(s) in which features of the kernel evolve. In addition, we aggregated the results for feature evolving

[12]The spreadsheets used during this experiment are available on our website: http://swerl.tudelft.nl/bin/view/NicolasDintzner/WebHome

in a single space, two spaces, and three variability spaces, with raw quantitative information and the percentage of those features (in *italic* in the table). For instance, in release 3.10, FEVER captured 5208 feature **TimeLines**. Among those, 654 evolved solely in the variability model (V.M.), and the total number of features that evolved through changes in a single space is 3407, or 78.19% of the evolving features in that release.

The table also presents the average and median number of features evolving in each combination of spaces over the studied period of time. We can see in the penultimate column of Table 4.10 that, on average over 15 releases, 4538 feature evolved and that, on average, only 7.43% of them evolved in all three spaces.

Release	3.10	3.11	3.12	3.13	3.14	3.15	3.16	3.17	3.18
Number of timelines	5208	4397	4424	4581	4503	4960	4099	4322	3797
V.M. only	654	508	909	357	390	608	462	487	337
Mapping only	11	9	3	15	11	7	8	3	15
Source only	3407	2859	2586	3387	3325	3292	2799	2862	2695
Single space	*4072*	*3376*	*3498*	*3759*	*3726*	*3907*	*3269*	*3352*	*3047*
Single space (%)	*78.19*	*76.78*	*79.07*	*82.06*	*82.74*	*78.77*	*79.75*	*77.56*	*80,25*
V.M. & mapping	39	22	14	20	19	15	21	33	14
V.M & source	632	549	588	453	442	522	451	450	366
source & mapping	54	67	56	68	65	59	76	71	50
Two spaces	*725*	*638*	*658*	*541*	*526*	*596*	*548*	*554*	*430*
Two spaces (%)	*13.92*	*14.51*	*14.87*	*11.81*	*11.68*	*12.02*	*13.37*	*12.82*	*11.32*
All spaces	411	383	268	281	251	457	282	416	320
All spaces (%)	*7.89*	*8.71*	*6.06*	*6.13*	*5.57*	*9.21*	*6.88*	*9.63*	*8.43*

Release	3.19	4.0	4.1	4.2	4.3	4.4	Average	Median
Number of timelines	5131	3432	4082	4316	4159	3967	4358	4322
V.M. only	355	330	337	406	387	330	547,1 (10.44%)	390 (9.40%)
Mapping only	3	23	11	29	1	5	10,27 (0.2%)	9 (0.20%)
Source only	3962	2369	2932	2954	2967	2884	3019 (69.27%)	2932 (69.03%)
Single space	*4320*	*2722*	*3280*	*3389*	*3355*	*3219*	*3486*	*3376*
Single space (%)	*84.19*	*79.31*	*80.35*	*78.52*	*80.69*	*81.14*	*79.96*	*79.75*
V.M. & mapping	9	8	17	21	14	14	18,67 (0.45%)	17 (0.41%)
V.M & source	428	349	415	462	449	410	467 (10.65%)	450 (79.75%)
source & mapping	60	63	86	91	56	71	66,2(1.51%)	65 (1.48%)
Two spaces	*497*	*420*	*518*	*574*	*519*	*495*	*549,3*	*541*
Two spaces (%)	*9.69*	*12.24*	*12.69*	*13.30*	*12.48*	*12.48*	*12.61*	*12.48*
All spaces	314	290	284	353	284	253	323,1	290
All spaces (%)	*6.12*	*8.45*	*6.96*	*8.18*	*6.83*	*6.38*	*7.43*	*6.95*

Table 4.10: Co-evolution of edited features over time. Values in italics are computed, while values in regular fonts are obtained using Neo4j queries.

Regarding the co-evolution of artefacts with respect to feature evolution, we can see that most features evolve only through their implementation.

On average and over the studied period of time, 69.27% of evolving features only changed in their implementation, either modification of the mapped artefact or modification of code blocks - #ifdef block. We can order the combination of spaces in which features are most likely to evolve as follows:

1. Source only (69.27%);

2. V.M. only (10.44%), and V.M. with Source (10.65%);

3. All three spaces (7.43%);

4. Any other combination of spaces occurs, on average over the studied period of time less than 2% of the time.

Table 4.11 show the evolution of all changed features, by spaces, over the entire studied period of time, i.e., 15 releases, approximately two years. The results show that, over the 15 releases, 4111 changed features among the 17448 features that were changed evolved in all three spaces. We can see that half of features (49.94%) evolved in a single space during that time.

Spaces	Count	Ratio (%)
Number of timelines	17448	100.00
V.M. only	1856	10.64
Mapping only	23	0.13
Source only	6835	39.17
Single space	*8714*	49.94
V.M. & mapping	214	1.23
V.M & source	4185	23.99
source & mapping	224	1.28
Two spaces	*4623*	*26.49*
All spaces	4111	23.56

Table 4.11: Co-evolution of edited features aggregated by feature, over the entire studied period of time. Values in italics are computed, while values in regular fonts are obtained using Neo4j queries.

Given our results, we can answer our third research question, RQ3: To what extent do artefacts in the different variability spaces co-evolve during the evolution of features?

> In a given release, a majority of features (79.96%) evolve without any co-evolution in the different variability spaces, their evolution occurs within a single space. The percentage of feature evolution performed by modification of multiple spaces is low (less than 25%) but remains relatively constant over the studied period of time.
>
> Over a longer period of time, more features will evolve in multiple spaces (50% after 15 releases).

7.3. RESULTS: CO-EVOLUTION AUTHORSHIP

Table 4.12 shows the spaces affected by authors commits in the Linux kernel. For each release, it presents the number of authors and the number of authors whose commits affected the different combinations of spaces. In release 3.18, among the 1507 authors, 12 committed changes modifying only the mapping space. In that same release, the number of authors whose changes modified only a single space is 1134, representing 78.9% of all authors.

The last two columns of the table show the average and median number of authors and the spaces they affected, with the aggregated values per spaces, over the studied period of time. We can see in the last column that the median number of authors in the studied releases is 1495, and the median number of authors who modified all spaces is 161, representing 10.98% of the authors.

Release	3.10	3.11	3.12	3.13	3.14	3.15	3.16	3.17	3.18
Number of authors	1433	1304	1362	1400	1481	1535	1513	1461	1507
V.M. only	8	6	7	10	12	11	7	5	6
Mapping only	9	7	8	5	7	18	8	21	12
Source only	1064	960	1071	1043	1101	1152	1163	1069	1116
Single space	*1081*	*973*	*1086*	*1058*	*1120*	*1181*	*1178*	*1095*	*1134*
Single space (%)	*77.55*	*77.04*	*80.03*	*78.60*	*77.99*	*79.58*	*80.03*	*77.66*	*78.59*
V.M. & mapping	2	0	2	1	0	0	0	1	4
V.M & source	77	63	78	83	81	63	70	84	75
source & mapping	64	62	52	61	74	77	73	63	70
Two spaces	*143*	*125*	*132*	*145*	*155*	*140*	*143*	*148*	*149*
Two spaces (%)	*10.26*	*9.90*	*9.73*	*10.77*	*10.79*	*9.43*	*9.71*	*10.50*	*10.33*
All spaces	170	165	139	143	161	163	151	167	160
All spaces (%)	*12.20*	*13.06*	*10.24*	*10.62*	*11.21*	*10.98*	*10.26*	*11.84*	*11.09*

Release	3.19	4.0	4.1	4.2	4.3	4.4	Average	Median
Number of authors	1495	1495	1576	1630	1607	1636	1495,67	1495,00
V.M. only	20	6	12	6	7	11	8,93 (0.62%)	7,00 (0.52%)
Mapping only	15	15	14	21	16	17	12,87 (0.88%)	14,00 (0.92%)
Source only	1085	1121	1181	1195	1251	1210	1118,6 (77.24%)	1116,00 (77.49%)
Single space	*1120*	*1142*	*1207*	*1222*	*1274*	*1238*	*1140.60*	*1134.00*
Single space (%)	*77.78*	*79.75*	*79.56*	*77.44*	*80.08*	*79.31*	*78.73*	*78.6*
V.M. & mapping	1	2	3	1	1	2	1,33 (0.09%)	1,00 (0.07%)
V.M & source	82	73	74	78	75	86	76,13 (5.27%)	77,00 (4.85%)
source & mapping	80	56	79	85	71	66	68,87 (4.75%)	70,00 (4.85%)
Two spaces	*163*	*131*	*156*	*164*	*147*	*154*	*146.33*	*147*
Two spaces (%)	*11.32*	*9.15*	*10.28*	*10.39*	*9.24*	*9.87*	*10.11*	*10.26*
All spaces	157	159	154	192	170	169	161.33	161.00
All spaces (%)	*10.90*	*11.10*	*10.15*	*12.17*	*10.69*	*10.83*	*11.16*	*10.98*

Table 4.12: Authorship of variability spaces over time. Values in italics are computed, while values in regular fonts are obtained using Neo4j queries

Regarding authorship of the different spaces, we can see that a majority of developers, over the course of a release, modified only the implementation space. In this context, this means that they touched the implementation of a feature (mapped artefact) or a code block (#ifdef block). Our results show that on average, over the studied period of time, this is true for 77.24% of authors. When authors touch multiple spaces, they are less likely to modify only the variability model and the mapping (0.09% of authors on average) than other combinations of spaces. Finally, between 10.2% and 13.06% of authors perform modifications spreading across all three spaces. We can see from Table 4.12 that this percentage varies very little over the studied period of time.

Table 4.13 present the authorship of variability spaces aggregated over the 15 releases we studied. The table shows that 17.47% of the 6645 authors we identified changed fea-

tures by editing all three variability spaces. Over the studied period of time, 72.47% of authors touched only a space, and a majority (71.33%) of authors focused solely on the source code.

Spaces	Count	Ratio (%)
Authors	6645	100.00
V.M. only	28	0.42
Mapping only	48	0.72
Source only	4740	71.33
Single space	*4816*	*72.47*
V.M. & mapping	4	0,06
V.M & source	316	4.76
source & mapping	318	4.79
Two spaces	*638*	*9.60*
All spaces	1191	17.92

Table 4.13: Authorship of variability spaces, aggregated by author over the entire studied period of time. Values in italics are computed, while values in regular fonts are obtained using Neo4j queries.

With those results, RQ4: To what extent are developers facing co-evolution over the course of a release?

> On a given release, only a minority (less than 25%) of developers will make changes to multiple spaces. The percentage of commit authors dealing with co-evolution is thus low, but stable over time. A majority of authors (approximately 75% in each release, and approximately 71% over time) will focus only on source code.

7.4. ON CO-EVOLUTION IN LINUX

In our experiments, we extracted feature-related changes from release v3.10 (June 2013) until release v4.4.(January 2016). The development of the kernel started much earlier than the first release studied in this work. Development practices in the Linux kernel are well documented and the development process can be considered as very mature. What we observe are changes occurring in a stream-lined development process. This in itself might explain the regularity in the data we gathered in terms of co-evolution and authors edits in various spaces. This regularity suggests that occurrences of co-evolution in feature evolution, or author experience of co-evolution will remain the same until the next upheaval of the development process or of the system's architecture.

With this in mind, we note that most developers did not perform changes in multiple spaces. Over time, a majority (71.33%) of developers only modified the implementation space. This is visible in our results, both when describing author's contributions to individual releases, and their contribution over the studied period of time. However, this does not mean that developers cannot introduce dead code blocks or false optional blocks in the implementation. Valid changes to the implementation do require some knowledge on feature support in all spaces. Developers still benefit from tools focusing on validation of the consistency of features across spaces. Such tools, such as Kbuild-

Miner (Nadi and Holt, 2012), TypeChef (Kenner et al., 2010), or Undertaker (Tartler et al., 2009), usually require the extraction of variability information from all variability spaces. Then they aggregate the information validate their consistency. If, as shown by our results, in most cases the VM and the mapping remain untouched, the information required from those artefacts to run consistency checks can be cached. According to our results, on a given release, more than 75% of authors could use this cached information - making cross-space variability more efficient. This would reduce the cost of variability consistency checks across spaces. Yet, for more complex change scenarios, a thorough and complete analysis is still required.

8. Threats to Validity

We present in this section the threats to validity of the two parts of our study: the change extraction process from developers' commits, and our exploratory study of co-evolution in the Linux kernel.

8.1. Threats to Validity: Feature-Oriented Change Extraction

Let us first discuss the limitations and threats to the validity of the FEVER change extraction process.

Limitations. Our evaluation shows that FEVER captures feature-related changes with a relatively high accuracy (87,2% of commits extracted completely correctly). However, FEVER does not capture all feature related information in all artefacts. Because FEVER operates on a file-basis, with a text-based parser, certain constructs in the variability model or the mapping are not captured. The limitations of FEVER for each space are mentioned in the relevant sub-sections of Section 3.

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 mappings. 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 change attributes our approach currently 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.

Using manual analysis for validation purposes is inherently fault prone. The difference in populations of changes observed between the initial and enhanced versions of the tool does highlight this. For this evaluation, the manual review of commits was performed twice - for the entire dataset, leaving a small time gap (between 2 days and a week) between the two evaluation rounds. While the errors identified in the initial evaluation lead to a significant update for some change attributes (namely "added feature references" in the code), evaluation errors occurred in less than 5% of the commits.

The evaluation of the new FEVER heuristics, compared to its previous version, highlights significant improvement of accuracy on specific change attributes. In particular the capture of code block changes with preserved code block improved from a precision

of 32% to 93.3%. Moreover, the added change attributes (namely artefact changes, additional artefact type, and **TimeLine** relationships to **FeatureEdit**) were captured with a good precision and recall (at least 80%).

Because the FEVER approach is based on heuristics, it is neither sound nor complete. But for more than 80% of the extracted commits, the data does reflect changes performed by developers. While this may be a limitation when searching for very specific changes, with specific change attributes, overall trends and statistics done over the course of a release reflect developer's activities on features in the Linux kernel with sufficient accuracy to draw conclusions from it.

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. The models used for comparison do contain attributes that are very tightly related to the technology used in the Linux kernel. However, there are several other systems using those very same technologies, such as aXTLs[13] and uClibc[14], on which our prototype - and thus our approach - would be directly applicable.

For other types of systems, one would need to adapt the model reconstruction phases depending on the system under study. If we consider another operating system such as eCos[15], one would need to rebuild the same change model from features described in the CDL language[16] instead of Kconfig. Concretely, this amounts to creating a CDL parser capable to build the same EMF variability model representation used in this work to initiate the comparison process. Attributes such as default value, select, or visibility would be relevant, and the "select" attribute can simply be left empty. A similar effort would be necessary to consider systems using the Gradle build system[17], rather than the Make system. 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. Moreover, our work shows that model-based differencing is a suitable approach to extract feature related changes from heterogeneous artefacts in large scale systems.

Our 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 mappings from the implementation itself at this time. We cannot extend this work to runtime variability analysis without further study.

[13] aXTLS: http://axtls.sourceforge.net/index.htm
[14] uClibc: https://uclibc.org/
[15] eCos: http://ecos.sourceware.org/
[16] CDL : http://ecos.sourceware.org/ecos/docs-3.0/cdl-guide/reference.html
[17] Gradle: https://gradle.org/

8.2. THREATS TO VALIDITY: CO-EVOLUTION OF ARTEFACTS IN THE LINUX KERNEL

We now consider the threats to the validity of our study of co-evolution of artefacts in feature evolution and authorship.

Internal validity. As mentioned in the previous section, the author names used in this experiment do contain aliases. A potential side effect is that more developers many in practice perform changes to multiple spaces and this might not be reported in our results. However, our manual analysis on a single release revealed that few names (less than 5%) could be identified as aliases. A more in-depth study might identify more aliases, but the manual analysis we did covered most name variations taken into account in studies focusing on such problems (Kouters et al., 2012). The remaining variations were not considered as they did not occur in our sample. Because the number of aliases we found was small, and the percentage of developers not experiencing co-evolution is very high, we do not think that the presence of aliases would lead to a very different conclusion.

As mentioned in the previous sections, the FEVER approach is not exact. As a result, we can expect the actual co-evolution of artefacts and the ratio of developers dealing with co-evolution challenges to be slightly different from what is reported in this paper. However, our conclusions rely on significant trends observed over time (70% of features evolved only through their implementation) and over a long period of time (15 releases). Therefor, we argue that our conclusions hold despite the lack of exactness of the FEVER prototype.

External validity. The Linux kernel has been under development for more than two decades. This system is mature and has a well defined development process. This is observable in the regularity of our results over the studied time period. For less mature systems, one could expect feature-oriented co-evolution of artefacts to be more prominent. This could be confirmed by applying the FEVER approach to the first releases of the Linux development or running a case-study on a newer system. Moreover, the ratio of co-evolution of artefacts for evolving features or the ratio of developers dealing with co-evolution in other systems may differ from what we observed in the Linux kernel. Nonetheless, we argue that our results are representative of co-evolution for a long-lived highly variable system developed by a large team (more than a thousand developers).

9. RELATED WORK

Variability implementation in highly-configurable systems has been extensively studied in the past (Thüm et al., 2014a). Our approach relies on extraction and consolidation of variability evolution across the different variability spaces. 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.

The evolution of variability models was studied in the past as a mean to obtain insights on the evolution of the system as a whole (Lotufo et al., 2010), or manage the impact of changes to the system's capabilities (Dintzner et al., 2015b; Heider et al., 2012b). In our previous work (Dintzner et al., 2015a), we introduced FMDiff, an approach to extract feature model changes, that inpsired us for the extraction of variability model

changes.

To capture the evolution of features, we need to track the evolution of their mapping. Studies focusing on co-evolution of artefacts (Neves et al., 2015; Passos et al., 2015) also place the mapping as a central element in the description of feature evolution. As shown by Adams et al. (Adams et al., 2008) in the Linux kernel, the build system evolves: the size and complexity of the build scripts increase over time, thus highlighting the relevance of build system evolution in the overall evolution of such highly configurable system.

Several studies present methods to extract variability information from build systems (Makefiles) (Dietrich et al., 2012a; Nadi et al., 2014; Zhou et al., 2015). Such approaches are designed to study the current state of the system, and rely on a complete description of the system. In this study, we took a different approach: FEVER focuses on changes performed on individual changed files. We developed a custom Makefile parser allowing us to extract information relying on modified artefacts only. Similarly to Nadi et al. (Nadi et al., 2014) and Dietrich et al. (Dietrich et al., 2012a) we rely on parsing rather than symbolic execution as was done by Zhou et al. (Zhou et al., 2015).

Variability implementations using annotative methods in source files were also studied in the past (Liebig et al., 2013), often for error detection (Kenner et al., 2010; Tartler et al., 2011, 2009). In this study, we used the approach presented in (Liebig et al., 2010) to identify code blocks and their conditions, and we then relied on this representation to build a model of implementation assets.

The variability model of a system, the mapping between features and assets, and variability support inside the implementation can all be supported by different technologies. In the eCos environment,[18] assets associated with features are directly included in the variability modeling language (CDL) specification. The Puppet[19] infrastructure offers a practical way of decoupling configuration and implementation (Sharma et al., 2016). Variability support at an implementation level can also be performed in a number of ways (Kästner and Apel, 2008). The FEVER approach does not encompass of possible ways of supporting variability in software system. However, this indicates that FEVER could be extended to be applied to a wide range of systems.

Only few studies focused on the co-evolution of artefacts in all three variability spaces: variability model (VM), mapping, and implementation. Neves et al. (Neves et al., 2015) 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. (Passos et al., 2015). 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. The approach proposed by Passos et al. relies on scripts to identify commits in which features in the Linux kernel are added and removed, and retrieve related information such as information regarding commits, name of the changed features, feature hierarchy, and the associated Linux release. From this initial information, extensive manual work is necessary to analyze changes of each type of artefacts, and their relationships. In comparison, the FEVER approach automatically extracts feature-related information from Kconfig file changes but also performed feature-related information extraction from other artefacts, such as Makefile and source files. While such in-

[18] CDL Language: http://ecos.sourceware.org/docs-1.3.1/cdl/language.properties.html
[19] Puppet : https://puppet.com/

formation was taken into account during the manual analysis performed in the context of (Passos et al., 2015), FEVER makes such information readily available. For instance, using FEVER, one can know using the extracted if a feature-change in a Makefile is related to a feature change in the Kconfig file.

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 (Begel et al., 2010). 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 (Passos and Czarnecki, 2014) 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 across the variability, mapping, and implementation space.

10. Conclusion and Research Directions

In this paper, we presented FEVER, an approach to automatically extract and build a feature-centered representation of changes in commits affecting the implementation of features in highly variable software 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 87.2% of the studied commits.

Our exploratory study of co-evolution in the Linux kernel showed that co-evolution of artefacts during feature evolution does occur, but, over a single release, most features only evolve through their implementation. A majority of developers focus only on the feature implementation and, over the course of a release, only few modify variability spaces beyond the implementation. We also found that, while co-evolution of artefacts occurs in every release, they account for less than 22% of feature evolution scenarios, and only 11% of authors will modify all variability spaces over the course of a release, but over time, 69,51% of authors will only modify the implementation of features without affecting the variability model or feature-asset mapping.

Through this work we make the following key contributions:

- a model-based approach to extract and consolidate feature changes across variability spaces
- an model of feature-oriented changes, focusing on the co-evolution of artefacts in different variability spaces during feature evolution
- an evaluation of the FEVER prototype implementation, as well as a evaluation of the improvement with respect to its previous installment
- a quantitative study describing the frequency of artefact co-evolution in the context of feature changes from a feature perspective, and authorship perspective

- several examples demonstrating the potential usage and value of the data gathered by such an approach for developers and researchers working on configurable software systems
- an implementation of FEVER as well as the full dataset, available for download[20]

There several ways in which the FEVER approach and its evaluation can be further enhanced in the future. First, let us consider potential improvement regarding the approach itself. At a variability model level, one could consider extracting semantic changes rather than syntactic changes as suggested by the work of Rothberg et al. (Rothberg et al., 2016). Efficient semantic differencing on a variability model as large as the Linux kernel V.M. is a challenging task. Moreover, given the potential size of a configuration of the Linux kernel, i.e., thousands of features, one would have to consider how to present this information in a way that can be useful to a human developer, making this an interesting research challenge. Regarding mapping changes, the current FEVER approach captures only change information contained within changed Makefiles. A more precise approach would be to capture the exact presence condition of assets, rather than the main features participating in that condition. Changes in the presence conditions will require a computationally intensive process, and the output might be difficult to interpret by a human. This is a direction we did not explore so far, but would be valuable to obtain a more sound and complete view on co-evolution changes in highly variable systems. On a source code level, FEVER does not consider file dependencies. A change to an #include statement could be a sign of changes in the relationships between features implemented within those files. Such changes are not necessarily represented in the mapping nor the variability model. A potential improvement of the FEVER approach would consist in taking into account file dependencies, and identify the nature of the symbols tying those files (functions, variable, type definitions, and so on).

To further evaluate the capabilities of the FEVER approach, we intend to apply FEVER to other systems. Candidates for such work would be systems relying on different technologies for variability model description and feature-asset mappings. The improved FEVER change meta-model and algorithm, as well as our observation on co-evolution open new exciting research directions. The information captured by FEVER on changed features could prove to be useful in the domain of test case selection for highly configurable systems. Combining the work of Vidacs et al. (Vidacs et al., 2015) on test selection in highly configurable software based on configuration and code coverage, and the work on of Soetens et al. (Soetens et al., 2016) on change-based test selection supported by FEVER data could lead to the discovery of efficient new techniques to support testing in the context of highly variable software.

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

5

Conclusion

And that is the way to get the greatest possible variety, but with all the order there could be; i.e. it is the way to get as much perfection as there could be.

G.W. Leibniz

Variant-rich software systems are characterized by a high complexity. Variation points in the implementation and build system, to derive many variants from a single code base, are an additional complexity compared to single variant systems. Such systems are usually accompanied by a formalized variability model. This additional information, required to derive valid configurations, must be maintained over time along side of other development artefacts. Throughout the evolution of the system, engineers must guarantee that the variability described by the variability model and the variation points within the implementation remain consistent. For those reasons, the evolution of variant-rich systems is more challenging than the evolution of single-variant systems.

In order to facilitate the development of such systems, through development tools or design approaches and techniques, we need an understanding of the evolution of such in real-life scenarios. Such information is paramount to determine which scenarios should be supported, and how this support may be achieved. In this thesis, our goal is to identify such scenarios in a real-world large-scale system, namely the Linux kernel, and to provide the means to gather such information.

5

1. SUMMARY OF THE CONTRIBUTIONS

The main contributions of this thesis can be summarized as follows:

- *A model-based approach to extract feature-related changes in heterogenous artefacts.* First designed to extract changes in a variability model between two releases, our approach evolved, based on recent work on complex evolution scenarios in variant-rich systems, to encompass feature changes in a number of related yet different artefacts. The approach allowed us to obtain and share a comprehensive view of feature-related changes in large-scale variant-rich software systems. This work started with the design of **FMDiff**, presented in Chapter 2, and is completed with **FEVER**, presented in Chapter 4.

- *An approach for multi-product line modelling for impact computation.* Motivated by an industrial problem, we devised a modelling approach allowing for simple impact computation during variability evolution. We showed that we were able to model a complex system from a set of different yet related specification documents, and assess the impact of feature changes on configurations of a system. We showed how, using such an approach we could detect impacts affecting the interfaces of the system. The approach and supporting tooling is presented in Chapter 3.

- *Empirical characterization of variability model evolution.* Through the approaches developed in our thesis, we provide empirical evidence that variability models evolve more from modifications of existing features than through feature additions or feature removals. We also showed that the sub-parts of a variability model tended to have different evolution scenarios, and therefore one should be careful when studying the evolution of a subset of features when attempting to describe the evolution of the complete system. The data supporting this contribution was gathered using **FMDiff** and **FEVER**, and is presented in Chapter 2 and Chapter 4.

- *A quantitative characterization of artefact co-evolution.* While we established that complex co-evolution scenarios do occur, through this work we showed that most features evolve solely through their implementation during maintenance operations. Similarly, we showed that a majority of developers do not face such complex scenarios during maintenance. While complex co-evolution scenarios should be supported, such scenarios, for features and authors alike, account for (approximately) 30% of changes. The data supporting this contribution was gathered using **FEVER**, and is presented in Chapter 4.

- *Tools and dataset.* Throughout this work, we produced a number of tools designed to extract changes from version control systems of variant-rich software systems. We gathered datasets that we used to describe and analyze variant-rich system evolution. All the tools we built, as well as the collected datasets, were made available. [1]

2. RESEARCH QUESTIONS REVISITED

The work presented in this thesis is aimed at answering the following research question.

> **Research question**: *In the context of a variant-rich system, how do features evolve and how is that evolution reflected in the co-evolution of its concrete artefacts?*

To answer this broad question, we broke it down into smaller questions. Let us see how the contributions of this work allowed us to answer those questions. Then, we reflect on the overall answer that we can provide to our main research question.

Research Question 1: *What are the operations commonly performed on features in a large-scale variability model?*

We answered this question in Chapters 2 and 4. The evolution operation that is the most frequent in the studied system (the Linux kernel) is the modification of existing features. We reached this conclusion through the observation of changes in the variability model of the system between two releases. Similarly, in Chapter 4, we found that most evolution scenarios (more than 60%) occur in a single space, and most likely the source code. Such changes are modifications to existing features and existing feature implementations.

Research Question 2: *How can feature-level information be leveraged to assist engineers during change impact analysis in variant-rich systems?*

In Chapter 3, we presented a feature-oriented impact analysis approach based on the multi-product line representation for a variant-rich system. With such a representation of the variability of the system, engineers are able to "simulate" a feature change,

[1] https://github.com/NZR/FEVER—Linux-tool
https://github.com/NZR/Software-Product-Line-Research
http://data.4tu.nl/repository/uuid:181933e0-a380-4411-84ea-74aec5724810

and observe the impact, in terms of available sub-system configurations, throughout the system. From a simple feature change, one can observe the potential impact on the external interfaces of the system for instance. Such feature-change information can assist engineers in identifying the potential impact of the change, as well as traceability in the ripple of the impact. With this information, engineers are in a better position to estimate impact before the change is implemented, and identify which subsystem is at fault when the change propagates in unwanted areas of the systems (such as stable external interfaces).

In Chapter 4, we extracted a large dataset pertaining to feature evolution in the Linux kernel. With that information, we were able to answer the following research questions.

Research Question 3: *What role does co-evolution play in the evolution of features of variant-rich systems?*

The collected data allowed us to observe that a majority of feature evolution scenarios - over the course of a release, were focused on a single variability space, i.e., features evolve solely through their representations in a single type of artefacts. Moreover, in the majority of the cases, this artefact is the implementation of the feature. While complex co-evolution scenarios do occur in practice, such scenarios are not the most common way in which features evolve in variant-rich systems.

Research Question 4: *To what extent are developers facing co-evolution over the course of a release?*

Similarly to our results on feature evolution scenarios, the study performed in Chapter 4 shows that, over the course of a release, a minority of developers modifies features in more than a single space. The majority of developers focuses on the implementation of the features and modifies only the source code. While developers may still face complex evolution tasks within the source code, due to variability annotations in such artefacts, the modification of feature mappings and feature declarations within the variability model are not part of their daily routine. Again, a number of developers does modify all variability spaces, however, those are not tasks performed by the average developer working on a variant-rich system.

Those answers allow us to reflect on our original research question:
In the context of a variant-rich system, how do features evolve and how is that evolution reflected in the co-evolution of its concrete artefacts?

In order to fully understand any evolution scenario involving features, knowledge of their complete implementation, across all relevant artefacts, is a necessity for developers. However, studies of co-evolution of artefacts, focusing on addition and removal of features hide the fact that most common changes are neither additions nor removals but modifications and the most frequent changes do not involve complex modifications to heterogeneous artefacts.

In addition and removal scenarios, the affected features should change in all relevant artefacts. This includes at least the variability model, for the declaration of the feature. The other spaces are impacted based on the desired behavior of the feature on the variants in which the feature may be included. Technically, the only addition and removal operations that would **not** be accompanied by co-evolution of artefacts are operations on *abstract* features (in the sense of (Thum et al., 2011)), and those are not frequent.

If we explore our results from a system-wide perspective, then co-evolution is paramount. Over each release of the Linux kernel, several hundreds of features are added and removed. Therefore, co-evolution scenarios are relevant when maintaining a variant-rich system as a whole. The evolution of the system is characterized by co-evolution of artefacts, on a feature basis.

However, if we consider the evolution of such systems from a feature-oriented perspective, co-evolution scenarios are occurring at least twice: at creation and deletion time. For the remainder of the feature's lifecycle, changes are performed mostly on a single artefact at a time. In such cases, we could consider that the spaces that are not modified constitute a framework restricting what changes will be considered valid for the modified space. Individual features share the following characteristic with single-variant system: most of their lifecycle is spent doing maintenance, with small well-targeted changes.

3. EVALUATION

In this section, we reflect on the evaluation of our work. We divide this section in two parts. First we focus on the methodology used for the evaluation of each individual experiement, then we reflect on the usage of the Linux kernel as main case study for our work.

3.1. EXPERIMENTAL RESULTS

The core of our work was conducted following the design science methodology. We built tools and prototypes allowing us to extract information to reflect on the studied systems. The methodology we used to extract changes from heterogeneous artefacts is based on artefact transformation and heuristics. Using such an approach, we know that the gathered data is neither sound nor complete. The use of heuristics, artefact transformations, and the intrinsic variability (in terms of structure) of the studied artefacts guarantees that there will be corner cases where our design will misinterpret the observed changes. To mitigate such effect, we conducted extensive manual reviews of the dataset collected during our experiments.

To implement our prototype, we worked in an iterative manner, using a small subset of changes as references. Once we reached a satisfying accuracy of this training set, we froze the design and performed an evaluation on a larger set of commits. For this evaluation, we randomly selected a subset of the collected data and compared the output of our prototype against what could be observed in commits performed by developers (ground truth). We expressed the accuracy of the approach in terms of the resulting precision and recall, above 85% for a large majority of tracked change attributes. Similarly, our evaluation showed that 87,2% of commits from our sample are extracted correctly. All manual analyses were performed in two independent steps to minimize human errors.

Regarding the work on change impact on variability models, the experiment was performed in an industrial context and the approach itself was evaluated in cooperation with the domain experts of our partner institution. We used a set of simple scenarios for the design of the prototype. However, the final evaluation was performed in presence of domain experts, where the output of the tool was compared to their knowledge of the system under study. On each occasion, the output of the tool matched the expectation of the experts with respect to the avalaible information.

3.2. LINUX AS A CASE STUDY

Most of our results and conclusions are based on observations made on the evolution of the Linux kernel. This raises the question of the generalizability of the work to other systems. We partially address this point in Chapters 2 and 4, but we reflect here on this question in the perspective of the entire body of work presented in this thesis.

Regarding the statistical information on the co-evolution of artefacts in variant-rich systems. Because our results were obtained by observing a single system (the Linux kernel), it would be unwise to generalize them without further study. There are no specific reasons why the percentage of developers facing complex change scenarios would be the same on any variant-rich system as it is in the context of the Linux kernel. As a matter of fact, we pointed out in Chapter 4 that, if we had considered a different time frame for our study, the results would have greatly varied (by including the first commit of the kernel, yielding a large number of variability model changes).

Regarding the change extraction approach. The FEVER approach we present in Chapter 4 is not applicable directly to all variant-rich systems. The prototype is designed to analyze changes in Kconfig files, Makefiles, and annotated C code using pre-processor annotations (#ifdefs). While this may be a strong limitation to the applicability of the approach, we would like to point out that several other systems use this set of technologies (such as aXTLS[2] or uClibc[3]). In this case, the existing implementation of the approach should be usable with minor modifications.

Regarding systems relying on other technologies, the existing prototype implementation should be adapted. However, the system should have certain characteristics. First, the variability model must be explicit. Then, the mapping between features and assets should be retrievable, and the fine-grained variability support should be performed in an annotative manner. As mentioned earlier, we rely on heuristics to relate changes from various artefacts together. In the context of the Linux kernel, a strict naming convention is in place. Therefore, that task could be accomplished with a high degree of accuracy using simple heuristics. If this information is not available, then existing approaches can be used to identify assets related to features, but such approaches are usually computationally intensive, and may not be compatible with large-scale analysis of changes.

In a general case, the complete variability model of the system may not be available. In the context of our work with Philips Healthcare, variability information was scattered in specification documents (i.e., variability was not the dominant decomposition dimension of the system). There was no explicit variability model, such as can be found in

[2] http://axtls.sourceforge.net
[3] https://uclibc.org/

Linux. Therefore, we know that our approaches are not necessarily applicable to arbitrary industrial systems.

4. FUTURE WORK

This work constitutes a step toward a better understanding and support of development activities in the context of variant-rich software systems. In order to facilitate such development activities, several additional research paths could be followed.

Beyond Linux. Our work on co-evolution of artefacts in variant-rich software systems is currently limited to the Linux kernel. It would be interesting to see if our observations hold for other systems, in other domains than operating systems. We believe that we would witness similar trends regarding pervalence of artefact co-evolution scenarios for features and authors in other large-scale mature variant-rich systems - i.e., systems for which variation point implementation techniques are well-defined, and that follows well-defined development practices. More experimentation may confirm this.

The Nature of Features. During this work, we were unable to use changes operated at a variability model level to predict changes at the implementation level. For instance, a new relationship between two features does not imply the creation of a dependency at a code level between the implementation of those two features. While this may be an intrinsic limitation, we believe that this could be a side-effect of the loose semantic of feature relationship and feature types. However, we noted that, in the Linux kernel, additional semantics were included as naming conventions of features. For instance, certain features are present in the model to describe exposed capabilities, e.g., features with the "HAVE_" prefix. Similarly, certain dependencies are used to represent operational constraints ("depends on EXPERIMENTAL"), while others represent code dependency ("depends on I2C"). We suggest here that making explicit the "implicit" semantics of the variability model, by refining feature relationships and feature types, might allow easier change impact analysis, and facilitate change comprehension. Further work would be necessary to classify feature "usages" and "nature" to determine whether those can be described, and possibly provide design and implementation methods adapted to such features.

Feature Design Approach. We argue that part of what makes the development of the Linux kernel successful, especially for managing variability, is the fact that feature declaration, mapping, and implementation are - in a large number of cases, all contained within the same folder. So, while a feature's full implementation is scattered among a number of artefacts, it remains relatively "local" when we consider the relative location of those artefacts in the Linux file tree. With such an organization of artefacts, developers can easily find artefacts related to a feature. To prove that this locality is easing development practices, we could observe changes occurring in features described fully locally, and those that are not. We can then compare this to error-inducing commits, and search for possible correlations between those two factors.

Testing and change validation. Fine-grained variability-related change information can be of use to target tests of variant-rich systems. The change information on all artefacts should facilitate the identification of change configurations, allowing targeted testing on changed components. Such approaches would be very valuable when the number of possible configurations is so large that they cannot be all validated - such as it is in the

Linux kernel.

5. FINAL WORDS

Seven years ago, the support of complex co-evolution scenarios in variant-rich systems was deemed lacking (Babar et al., 2010). In this thesis, we make a step forward toward a better support of evolution operation performed by developers on variant-rich systems. While we do not provide the ultimate solution to variant-rich system development, it is our hope that, based on the contributions of this work, better evolution support tools may be developed, allowing developers to build more flexible designs with retaining the highest possible quality. The road ahead is still long, and the design and implementation of variant-rich systems and composable software components will remain a challenge for the next few years, but hopefully, this thesis constitutes a significant step in the right direction.

BIBLIOGRAPHY

Abal, I., Brabrand, C., and Wasowski, A. (2014). 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. ACM.

Abran, A., Bourque, P., Dupuis, R., and Moore, J. (2014). *Guide to the software engineering body of knowledge*. Number 3.0. IEEE Computer Society.

Acher, M., Collet, P., Lahire, P., and France, R. (2010a). Comparing Approaches to Implement Feature Model Composition. In Kühne, T., Selic, B., Gervais, M.-P., and Terrier, F., editors, *Proc. of the 6th European Conf. on Modelling Foundations and Applications*, ECMFA '10, pages 3–19. Springer Berlin Heidelberg.

Acher, M., Collet, P., Lahire, P., and France, R. (2010b). Managing multiple software product lines using merging techniques. Research report ISRN I3S/RR20 10 - 06 FR, Laboratoire dInformatique de Signaux et Systèmes de Sophia Antipolis - UNSA-CNRS.

Acher, M., Collet, P., Lahire, P., and France, R. (2010c). Managing Variability in Workflow with Feature Model Composition Operators. In Baudry, B. and Wohlstadter, E., editors, *Proc. of the 9th International Conf. on Software Composition*, SC '10, pages 17–33. Springer Berlin Heidelberg.

Adams, B., Schutter, K. D., Tromp, H., and Meuter, W. D. (2008). The Evolution of the Linux Build System. *Electronic Communications of the EASST*, 8(0).

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

Babar, M. A., Lianping Chen, and Shull, F. (2010). Managing Variability in Software Product Lines. *IEEE Software*, 27(3):89–91, 94.

Bagheri, E. and Gasevic, D. (2011). Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612.

Batory, D. (2004). Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 702–703. IEEE Computer Society.

Batory, D., Barnett, J., Garza, J., Smith, K., Tsukuda, K., Twichell, B., and Wise, T. (1988). GENESIS: an extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1730.

Becan, G. (2013). Reverse Engineering Feature Models in the Real.

Begel, A., Phang, K. Y., and Zimmermann, T. (2010). Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1 of *ICSE '10*, page 125. ACM Press.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.

Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2005). Automated Reasoning on Feature Models. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Pastor, O., and Falcão e Cunha, J., editors, *Advanced Information Systems Engineering*, volume 3520, pages 491–503. Springer Berlin Heidelberg, Berlin, Heidelberg.

Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., and Czarnecki, K. (2015). What is a feature?: a qualitative study of features in industrial software product lines. pages 16–25. ACM Press.

Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., and Wasowski, A. (2013a). A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, page 1. ACM Press.

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. (2010). Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the international conference on Automated software engineering*, ASE '10, page 73. ACM Press.

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. (2013b). A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640.

Binkley, D., Capellini, R., Raszewski, L., and Smith, C. (2001). An implementation of and experiment with semantic differencing. pages 82–91. IEEE Comput. Soc.

Bird, C., Pattison, D., D'Souza, R., Filkov, V., and Devanbu, P. (2008). Latent Social Structure in Open Source Projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 24–35, New York, NY, USA. ACM.

Botterweck, G., Pleuss, A., Dhungana, D., Polzer, A., and Kowalewski, S. (2010). 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.

Breivold, H. P., Crnkovic, I., and Eriksson, P. J. (2008). Analyzing software evolvability. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 327–330.

Bruntink, M., van Deursen, A., D'Hondt, M., and Tourwé, T. (2007). Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. page 199. ACM Press.

Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., and Raskin, J.-F. (2013). Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089.

Clements, P. and Northorp, L. (2002). *Software Product Lines, 2nd edition*. Addison-Weasley, Reading, MA.

Clements, P. C., Jones, L. G., McGregor, J. D., and Northrop, L. M. (2006). Getting there from here: a roadmap for software product line adoption. *Communications of the ACM*, 49(12):33.

Czarnecki, K. and Antkiewicz, M. (2005). Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Glück, R. and Lowry, M., editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer Berlin / Heidelberg.

Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U. (2002). Generative Programming for Embedded Software: An Industrial Experience Report. In Batory, D., Consel, C., and Taha, W., editors, *Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 156–172. Springer Berlin Heidelberg.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2005). Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169.

Dietrich, C., Tartler, R., Schröder-Preikschat, W., and Lohmann, D. (2012a). 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.

Dietrich, C., Tartler, R., Schröder-Preikshat, W., and Lohmann, D. (2012b). Understanding Linux feature distribution. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, MISS'12, pages 15–20. ACM.

Dintzner, N., Deursen, A. v., and Pinzger, M. (2015a). Analysing the Linux kernel feature model changes using FMDiff. *Software & Systems Modeling*, pages 1–22.

Dintzner, N., Kulesza, U., van Deursen, A., and Pinzger, M. (2015b). Evaluating Feature Change Impact on Multi-product Line Configurations Using Partial Information. In Schaefer, I. and Stamelos, I., editors, *Proceedings of the 14th Conference on Software Reuse*, ICSR '15, pages 1–16. Springer International Publishing.

Dintzner, N., Van Deursen, A., and Pinzger, M. (2013). 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.

Dintzner, N., van Deursen, A., and Pinzger, M. (2016). FEVER: Extracting Feature-oriented Changes from Commits. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 85–96, New York, NY, USA. ACM.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95.

Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210 – 224.

Fahrenberg, U., Legay, A., and Wsowski, A. (2011). Vision Paper: Make a Difference! (Semantically). In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Whittle, J., Clark, T., and Kühne, T., editors, *Model Driven Engineering Languages and Systems*, volume 6981, pages 490–500. Springer Berlin Heidelberg, Berlin, Heidelberg.

Fogdal, T., Scherrebeck, H., Kuusela, J., Becker, M., and Zhang, B. (2016). Ten years of product line engineering at Danfoss: lessons learned and way ahead. pages 252–261. ACM Press.

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

Gheyi, R., Massoni, T., and Borba, P. (2006). A theory for feature models in alloy. In *First alloy workshop*, Allow Workshop, pages 71–80.

Giger, E., Pinzger, M., and Gall, H. (2012). 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.

Giger, E., Pinzger, M., and Gall, H. C. (2011). Comparing Fine-grained Source Code Changes and Code Churn for Bug Prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, New York, NY, USA. ACM.

Godfrey, M. and German, D. (2008). The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 129–138.

Guo, J. and Wang, Y. (2010). Towards Consistent Evolution of Feature Models. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 451–455, Berlin, Heidelberg. Springer-Verlag.

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

Hartmann, H. and Trew, T. (2008). Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *Proc. of the 12th International Software Product Line Conference*, SPLC '08, pages 12–21. IEEE.

Heider, W., Rabiser, R., Grünbacher, P., and Lettner, D. (2012a). Using Regression Testing to Analyze the Impact of Changes to Variability Models on Products. In *Proc. of the 16th International Software Product Line Conference*, SPLC '12, pages 196–205. ACM.

Heider, W., Vierhauser, M., Lettner, D., and Grunbacher, P. (2012b). 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, pages 1 –10.

Hellebrand, R., Silva, A., Becker, M., Zhang, B., Sierszecki, K., and Savolainen, J. (2014). 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. ACM.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Q.*, 28(1):75–105.

Holl, G., Thaller, D., Grünbacher, P., and Elsner, C. (2012). Managing Emerging Configuration Dependencies in Multi Product Lines. In *Proc. of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 3–10. ACM.

Hubaux, A., Xiong, Y., and Czarnecki, K. (2012). A user survey of configuration challenges in Linux and eCos. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 149–155, New York, NY, USA. ACM.

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., LeSSenich, O., Becker, M., and Apel, S. (2015). Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*.

Hölzl, F. and Feilkas, M. (2010). 13 AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In Giese, H., Karsai, G., Lee, E., Rumpe, B., and Schätz, B., editors, *Model-Based Engineering of Embedded Real-Time Systems*, number 6100 in Lecture Notes in Computer Science, pages 317–322. Springer Berlin Heidelberg.

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

Juanjuan Jiang, Ruokonen, A., and Systa, T. (2005). Pattern-based variability management in Web service development. page 12 pp. IEEE.

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

Kenner, A., Kästner, C., Haase, S., and Leich, T. (2010). 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. ACM.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Goos, G., Hartmanis, J., van Leeuwen, J., Akit, M., and Matsuoka, S., editors, *ECOOP'97 Object-Oriented Programming*, volume 1241, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kouters, E., Vasilescu, B., Serebrenik, A., and Brand, M. G. J. v. d. (2012). Who's who in Gnome: Using LSA to merge software repository identities. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 592–595.

Koziolek, H., Goldschmidt, T., de Gooijer, T., Domis, D., Sehestedt, S., Gamer, T., and Aleksy, M. (2016). Assessing software product line potential: an exploratory industrial case study. *Empirical Software Engineering*, 21(2):411–448.

Krueger, C. (2006). New methods in software product line development. In *Proc. of the 10th International Software Product Line Conference*, SPLC '06, pages 95–99. IEEE Computer Society.

Kumara, I., Han, J., Colman, A., Nguyen, T., and Kapuruge, M. (2013). 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, pages 567–574. IEEE.

Kästner, C. and Apel, S. (2008). Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, Proceedings*, pages 35–40. Department of Informatics and Mathematics, University of Passau.

Lehman, M. (1996). Laws of software evolution revisited. *Software process technology*, pages 108–124.

Lemley, M. A. and O'Brien, D. W. (1997). Encouraging Software Reuse. *Stanford Law Review*, 49(2):255.

Lengauer, C., Apel, S., Bolten, M., GröSSlinger, A., Hannig, F., Köstler, H., Rüde, U., Teich, J., Grebhahn, A., Kronawitter, S., Kuckuk, S., Rittich, H., and Schmitt, C. (2014). ExaStencils: Advanced Stencil-Code Engineering. In Lopes, L., ilinskas, J., Costan, A., Cascella, R. G., Kecskemeti, G., Jeannot, E., Cannataro, M., Ricci, L., Benkner, S., Petit, S., Scarano, V., Gracia, J., Hunold, S., Scott, S. L., Lankes, S., Lengauer, C., Carretero, J., Breitbart, J., and Alexander, M., editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806, pages 553–564. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-14313-2_47.

Li, L., Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., and Traon, Y. L. (2016). Mining families of android applications for extractive SPL adoption. pages 271–275. ACM Press.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering*, volume 1 of *ICSE '10*, page 105. ACM Press.

Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., and Lengauer, C. (2013). Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, page 81. ACM Press.

Lim, W. (1994). Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30.

Lotufo, R., She, S., Berger, T., Czarnecki, K., and Wasowski, A. (2010). Evolution of the Linux Kernel Variability Model. In Bosch, J. and Lee, J., editors, *Software Product Lines: Going Beyond*, number 6287 in Lecture Notes in Computer Science, pages 136–150. Springer Berlin Heidelberg.

Maoz, S., Ringert, J. O., and Rumpe, B. (2011a). ADDiff: semantic differencing for activity diagrams. page 179. ACM Press.

Maoz, S., Ringert, J. O., and Rumpe, B. (2011b). CDDiff: Semantic Differencing for Class Diagrams. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., and Mezini, M., editors, *ECOOP 2011 Object-Oriented Programming*, volume 6813, pages 230–254. Springer Berlin Heidelberg, Berlin, Heidelberg.

Matter, D., Kuhn, A., and Nierstrasz, O. (2009). Assigning bug reports using a vocabulary-based expertise model of developers. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140.

Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, New York, NY, USA. ACM.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in Software Evolution. In *International Workshop on Principles of Software Evolution*, pages 13–22. IEEE.

Mietzner, R., Metzger, A., Leymann, F., and Pohl, K. (2009). Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. pages 18–25. IEEE.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2014). Mining configuration constraints: static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 140–151. ACM Press.

Nadi, S. and Holt, R. (2012). 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.

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

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

Ommering, R. C. v. and Bosch, J. (2002). Widening the Scope of Software Product Lines - From Variation to Composition. In *Proc. of the 2nd International Conference on Software Product Lines*, SPLC '02, pages 328–347. Springer-Verlag.

Parnas, D. (1976). On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9.

Paskevicius, P., Damasevicius, R., and tuikys, V. (2012). Change Impact Analysis of Feature Models. In Skersys, T., Butleris, R., and Butkiene, R., editors, *Information and Software Technologies*, number 319 in Communications in Computer and Information Science, pages 108–122. Springer Berlin Heidelberg.

Passos, L. and Czarnecki, K. (2014). 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. ACM.

Passos, L., Czarnecki, K., Apel, S., Wasowski, A., Kästner, C., and Guo, J. (2013). 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. ACM.

Passos, L., Czarnecki, K., and Wasowski, A. (2012). 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.

Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wasowski, A., Czarnecki, K., Borba, P., and Guo, J. (2015). Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, pages 1–50.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag.

Potts, C. (1993). Software engineering research revisited. *IEEE Software*, 10(5):19–28.

Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. In Goos, G., Hartmanis, J., van Leeuwen, J., Akit, M., and Matsuoka, S., editors, *ECOOP'97 Object-Oriented Programming*, volume 1241, pages 419–443. Springer Berlin Heidelberg, Berlin, Heidelberg.

Queiroz, R., Passos, L., Valente, M. T., Apel, S., and Czarnecki, K. (2014). Does feature scattering follow power-law distributions?: an investigation of five pre-processor-based systems. In *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*, FOSD '14, pages 23–29. ACM Press.

Rabiser, D., Grunbacher, P., Prahofer, H., and Angerer, F. (2016). A prototype-based approach for managing clones in clone-and-own product lines. pages 35–44. ACM Press.

Reiser, M.-O. and Weber, M. (2007). Multi-level Feature Trees: A Pragmatic Approach to Managing Highly Complex Product Families. *Requirements Engineering*, 12(2):57–75.

Romano, D. and Pinzger, M. (2012). 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.

Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O., and Saake, G. (2008). 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.

Rothberg, V., Dintzner, N., Ziegler, A., and Lohmann, D. (2016). Feature Models in Linux: From Symbols to Semantics. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, pages 65–72, New York, NY, USA. ACM.

Rowe, D. and Leaney, J. (1997). Evaluating evolvability of computer based systems architectures-an ontological approach. In *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, pages 360–367.

Sampaio, G., Borba, P., and Teixeira, L. (2016). Partially Safe Evolution of Software Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference (to appear)*. ACM.

Schirmeier, H. and Spinczyk, O. (2009). Challenges in Software Product Line Composition. In *Proc. of the 42nd Hawaii International Conference on System Sciences*, HICSS '09. IEEE.

Schmid, K. (2013). Variability support for variability-rich software ecosystems. pages 5–8. IEEE.

Schobbens, P.-Y., Heymans, P., and Trigaux, J.-C. (2006). Feature Diagrams: A Survey and a Formal Semantics. pages 139–148. IEEE.

Schröter, R., Siegmund, N., and Thüm, T. (2013). Towards Modular Analysis of Multi Product Lines. In *Proc. of the 17th International Software Product Line Conference Co-located Workshops*, pages 96–99. ACM.

Seidl, C., Heidenreich, F., and ASSmann, U. (2012). Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference*, SPLC '12, pages 76–85. ACM.

Sharma, T., Fragkoulis, M., and Spinellis, D. (2016). Does Your Configuration Code Smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 189–200, New York, NY, USA. ACM.

She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K. (2010). The Variability Model of The Linux Kernel. In *Proceedings of the 4th International Workshop on Variability Modeling of Software-intensive Systems*, VaMoS '10, pages 45–51.

She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461 –470.

Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P. G., Apel, S., and Kolesnikov, S. S. (2013). Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. *Information and Software Technology*, (0).

Sincero, J., Schirmeier, H., Schröder-Preikschat, W., and Spinczyk, O. (2007). 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.

Sincero, J. and Schröder-Preikschat, W. (2008). The linux kernel configurator as a feature modeling tool. *SPLC*, pages 257–260.

Sincero, J., Tartler, R., Egger, C., Schröder-Preikschat, W., and Lohmann, D. (2010a). Facing the Linux 8000 Feature Nightmare. In SIGOPS, A., editor, *Proceedings of ACM European Conference on Computer Systems*, EuroSys '10, Paris, France.

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

Siy, H. P. and Perry, D. E. (1998). 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.

Soetens, Q. D., Demeyer, S., Zaidman, A., and Pérez, J. (2016). Change-based test selection: an empirical evaluation. *Empirical Software Engineering*, 21(5):1990–2032.

Spencer, H. and Collyer, G. (1992). *#ifdef Considered Harmful, or Portability Experience With C News*.

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

Tartler, R., Lohmann, D., Sincero, J., and Schröder-Preikschat, W. (2011). 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.

Tartler, R., Sincero, J., Schröder-Preikschat, W., and Lohmann, D. (2009). 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.

Thuem, T., Batory, D., and Kaestner, C. (2009). Reasoning about edits to feature models. In *Proc. of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264. IEEE Computer Society.

Thum, T., Kastner, C., Erdweg, S., and Siegmund, N. (2011). Abstract Features in Feature Modeling. In *Proceedings of the 15th International Software Product Line Conference*, SPLC '11, pages 191 –200.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014a). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45.

Thüm, T., Kaestner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014b). FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79(0):70–85.

Tian, Y., Lawall, J., and Lo, D. (2012). Identifying Linux Bug Fixing Patches. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 386–396. IEEE Press.

van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 45 –54.

Vidacs, L., Horvath, F., Mihalicza, J., Vancsics, B., and Beszedes, A. (2015). Supporting software product line testing by optimizing code configuration coverage. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–7.

White, J., Benavides, D., Schmidt, D. C., Trinidad, P., Dougherty, B., and Ruiz-Cortes, A. (2010). Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094–1107.

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

CURRICULUM VITÆ

Nicolas DINTZNER

07-11-1982	Born in Paris, France.

EDUCATION

2000	French School Vincent van Gogh (The Hague, Netherlands) High School Diploma
2001-2006	E.P.F (Sceaux, France) Generalist Engineering School Specialization in System Architecture
2006-2012	Dassault Systemes (Velizy, France) Software engineer
2012-2017	Technical University of Delft Netherlands (Delft, Netherlands) Ph.D in Software Engineering

Thesis:	Feature-Oriented Software Evolution
Promotor:	Prof. dr. A. van Deursen
Promotor:	Prof. dr. M. Pinzger

LIST OF PUBLICATIONS

PEER-REVIEWED PUBLICATIONS AS MAIN AUTHOR

6. **N. Dintzner**, A. van Deursen, and M. Pinzger, "FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems", in *Empirical Software Engineering* (EMSE), 2016, submitted, under review

5. **N. Dintzner**, A. van Deursen, and M. Pinzger, FEVER: Extracting Feature-oriented Changes from Commits, in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 8596.

4. **N. Dintzner**, Safe evolution patterns for software product lines, in *Proceedings of the 37th International Conference on Software Engineering*, Volume 2 (Doctoral Symposium paper, published in ICSE'15 companion proceedings), Pages 875-878.

3. **N. Dintzner**, U. Kulesza, A. van Deursen, and M. Pinzger, Evaluating Feature Change Impact on Multi-product Line Configurations Using Partial Information, in *Proceedings of the 14th Conference on Software Reuse*, 2015, pp. 116.

2. **N. Dintzner**, A. van Deursen, and M. Pinzger, Analysing the Linux kernel feature model changes using FMDiff", in *International Journal on Software and Systems Modeling*, pp. 122, May 2015.

1. **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*, 2013.

PEER-REVIEWED PUBLICATIONS AS CO-AUTHOR

2. V. Rothberg, **N. Dintzner**, A. Ziegler, and D. Lohmann, Feature Models in Linux: From Symbols to Semantics, in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, 2016, pp. 6572.

1. L. Passos, L. Teixera, **N. Dintzner**, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, J. Guo Coevolution of variability models and related software artifacts: A fresh look at evolution patterns in the Linux kernel, in *Empirical Software Engineering* (EMSE), May 2015.

SUPERVISED STUDIES

- Master thesis : "Architecture variability and multi-criteria optimization" by Jianfeng Zhao, in collaboration with the Technical University of Eindhoven and Philips Healthcare