Towards Efficient Analysis of Variation in Time and Space

Thomas Thüm
TU Braunschweig
Brunswick, Germany

Leopoldo Teixeira
Federal University of Pernambuco
Recife, Brazil

Klaus Schmid
University of Hildesheim
Hildesheim, Germany

Eric Walkingshaw
Oregon State University
Corvallis, USA

Mukelabai Mukelabai
Chalmers | University of Gothenburg
Gothenburg, Sweden

Mahsa Varshosaz
IT University of Copenhagen
Copenhagen, Denmark

Goetz Botterweck
Lero, University of Limerick
Limerick, Ireland

Ina Schaefer
TU Braunschweig
Brunswick, Germany

Timo Kehrer
Humboldt University of Berlin
Berlin, Germany

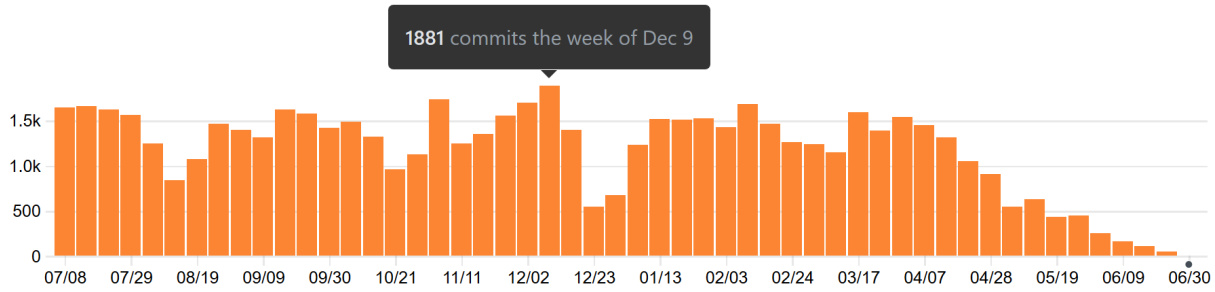

Figure 1: Variation in time: The Linux master branch contains 61,261 revisions (i.e., commits) between July 2018 and June 2019.

ABSTRACT

Variation is central to today's software development. There are two fundamental dimensions to variation: Variation in time refers to the fact that software exists in numerous revisions that typically replace each other (i.e., a newer version supersedes an older one). Variation in space refers to differences among variants that are designed to coexist in parallel. There are numerous analyses to cope with variation in space (i.e., product-line analyses) and others that cope with variation in time (i.e., regression analyses). The goal of this work is to discuss to which extent product-line analyses can be applied to revisions and, conversely, where regression analyses can be applied to variants. In addition, we discuss challenges related to the combination of product-line and regression analyses. The overall goal is to increase the efficiency of analyses by exploiting the inherent commonality between variants and revisions.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software verification and validation**; **Software evolution**.

KEYWORDS

software configuration management, regression analysis, software product lines, software evolution, software variation, variability management, product-line analysis, variability-aware analysis

ACM Reference Format:
Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3307630.3342414

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-6668-7/19/09…$15.00
https://doi.org/10.1145/3307630.3342414

1 INTRODUCTION

Software development is challenged by two dimensions of variability. The continuous development and improvement of software leads to numerous revisions of the software, which are supposed to replace each other. We refer to revisions as *variability in time*. Due to the high frequency of iterations it is often not feasible to completely analyze each revision and it would involve redundant effort as subsequent revisions are almost identical for large software systems. For instance, the Linux kernel is developed by hundreds of developers giving rise to 61,261 commits within the last year and a peak of 1881 commits in a single week, whereas there are even more revisions which did not make it into the master branch (cf. Figure 1, data accessed on July 1st, 2019). In peak weeks, every five minutes a new revision needs to be analyzed for the master branch (i.e., compiled and tested during continuous integration).

Besides variability in time, software is often developed in different variants that are designed to co-exist simultaneously. We refer to those software variants as *variability in space*. There are many reasons for the development of software variants, such as alternative hardware, conflicting requirements, or the optimization of non-functional properties. While for a low number of variants clone-and-own may be used, many variants are developed with dedicated implementation techniques in a product line [6, 14]. Product lines enable to generate software variants (a.k.a. products) for a selection of features. For instance, Linux has about 18,000 features which are mapped to implementation artifacts by means of the C preprocessor. While the exact number of features depends on the architecture and revision, the number of variants grows exponentially with the number of features [47]. To the best of our knowledge, the exact number of variants is not even known for Linux. However, it is consensus in the product-line community that it is not feasible to analyze them all separately [1, 17, 21, 22, 25, 29, 41, 46].

The massive variation imposed by revisions and variants requires efficient analysis techniques [50]. We recognize that analyses for revisions and variants have been proposed by largely different communities. We refer to analyses that have been designed to efficiently analyze revisions as regression analyses. Examples for regression analyses are regression testing [54], change impact analysis [9, 24], incremental program analysis [7, 13, 45], and regression verification [19, 23]. In contrast, analyses devoted to the analysis of variants are known as product-line analyses [47, 53]. Our goal is to bridge the gap between those communities by systematically discussing how regression analyses can be applied to variants and product-line analyses to revisions. Furthermore, we discuss how variation according to both dimensions, namely time and space, can be efficiently analyzed. We refer to such analyses as *product-line regression analyses*. Figure 2 gives an overview on the systematic behind our discussions and illustrates the structure of this paper. Overall, we make the following contributions:

- We provide a motivating example that illustrates the need for efficient analysis of variants and revisions (Section 2).
- We discuss the application of techniques in both directions, that is, the application of product-line analyses to revisions (Section 3) and the application of regression analyses to variants (Section 4).
- We discuss how product-line analyses and regression analyses can be applied to both dimensions of variation (Section 5).
- Finally, we provide directions for future work that is needed to overcome the identified challenges (Section 6).

2 MOTIVATING EXAMPLE

As a motivating example, we consider a single software product with functionality to store multiple objects in a list. The product p_1 is implemented by a single class called `Store`, as shown in Figure 3. The initial version implements a `read` method that only returns one object—the first object in the list. Later, this implementation is extended by another method to read all objects in the list:

```
public Object[] readAll() { return values; }      (1)
```

This change in the product's implementation constitutes the evolution of the product to a new revision p_1'; we refer to this

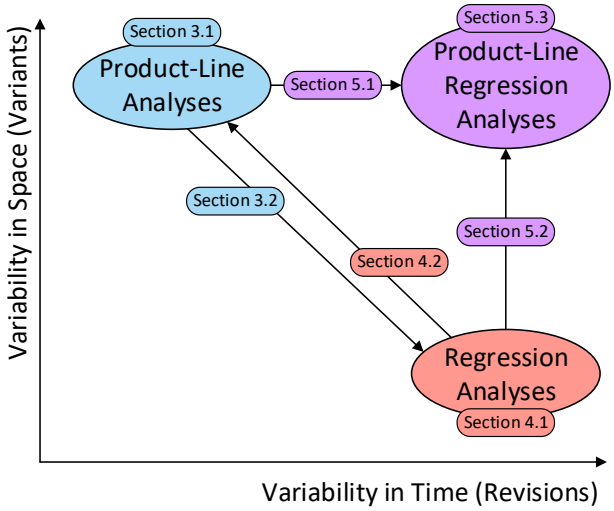

Figure 2: Efficient analyses for two dimensions of variability

evolution as *variation in time*, which is illustrated by the x-axis in Figure 2. Revisions are typically managed by a version control system [12], such as Git or Subversion. In this small example it would be feasible to compile the new revision again from scratch, but for large systems, such as Linux, this compilation can take hours. In the example, we could avoid some checks for the unchanged methods `read` and `set` by means of incremental compilation.

While product p_1' is supposed to replace product p_1, there are also cases where products are intended to co-exist in parallel. In addition to the existing multi-storage functionality, we may need support for single storage (which is less costly) and provide better security to the storage system by introducing access control. We refer to this kind of variation as *variation in space*, as illustrated by the y-axis in Figure 2. To manage variation in space, version control systems often do not scale considering that they are designed to support revisions (variation in time) rather than variants (variation in space). Although branches can be used to handle variation in space to a limited extent [6], the number of required branches can grow exponentially with the number of features due to the combinatorial explosion of possible feature combinations.

With software product-line engineering, product variants are automatically generated for a selection of features [6, 14]. For that

```
1  class Store {
2    private LinkedList values = new LinkedList();
3    Object read() {
4      return values.getFirst();
5    }
6    void set(Object value) {
7      values.addFirst(value);
8    }
9  }
```

Figure 3: A store for multiple objects (product p_1) [47]

```
1  class Store {
2  #IFDEF SingleStore
3    private Object value;
4  #ELSE
5    private LinkedList values =
6      new LinkedList();
7  #ENDIF
8  #IFDEF AccessControl
9    boolean sealed = false;
10 #ENDIF
11   public Object read() {
12 #IFDEF AccessControl
13     if (sealed)
14       throw new RuntimeException(
15         "Access denied!");
16 #ENDIF
17 #IFDEF SingleStore
18     return value;
19 #ELSE
20     return values.getFirst();
21 #ENDIF
22   }
23   public void set(Object value) {
24 #IFDEF AccessControl
25     if (sealed)
26       throw new RuntimeException(
27         "Access denied!");
28 #ENDIF
29 #IFDEF SingleStore
30     this.value = value;
31 #ELSE
32     values.addFirst(value);
33 #ENDIF
34   }
35 #IFDEF MultiStore
36   public Object[] readAll {
37     return values;
38   }
39 #ENDIF
40 }
```

Figure 4: An object store product line implemented with preprocessor annotations.

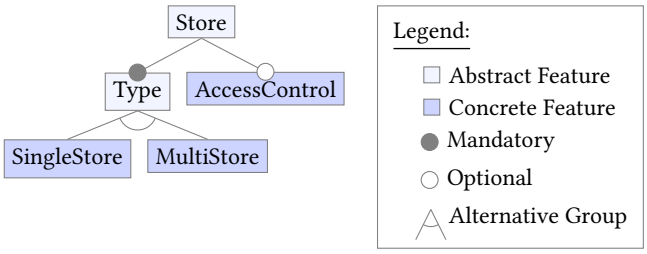

Figure 5: Feature model of the object store product line [47]

to work, valid feature combinations need to be specified and features need to be mapped to code artifacts [6]. The available features and their valid combinations are typically specified by means of feature models as illustrated in Figure 5. Our example has two mandatory features that are mutually exclusive—*SingleStore* and *MultiStore*—any valid product must have one of them, but not both. In addition, we have an optional feature called *AccessControl* that may or may not be included in a product. The feature model specifies a total of four valid configurations each defined by a set of selected features: $c_1 = \{MultiStore\}$, $c_2 = \{SingleStore\}$, $c_3 = \{MultiStore, AccessControl\}$, and $c_4 = \{SingleStore, AccessControl\}$. Figure 4 exemplifies how features can be mapped to code artifacts with conditional compilation. A preprocessor can remove parts of the code prior to compilation based on the selected features. For instance, product p_1' discussed above can be derived automatically by the preprocessor for the configuration c_1.

While our example product portfolio evolves to add new variants in space, variants may also be revised to improve their functionality, resulting in revisions of variants. For instance, the change from product p_1 to p_1' given in (1) may also be applied to an initial revision of the product line and result in adding the lines 35–39 of Figure 4. As the reader may have noticed, we introduced a type error by letting method `readAll` return a list of type `LinkedList` instead of the specified return type `Object[]`. A further revised implementation of the `readAll` method could be:

```
public Object[] readAll() { return values.toArray(); }   (2)
```

This would consequently lead to new revisions of variants derived for the configurations c_1 and c_3 since they contain feature *MultiStore*. Even though the compilation error is resolved with this revision of the product line, unit testing could uncover a security problem with method `readAll`, as it grants access to sealed object stores. Copying lines 12–16 to the beginning of method `readAll` would fix that problem, but results in a further revision of the product line.

The detection of the compilation error and the security problem is not straightforward for a product line. A particular challenge is that we cannot simply compile and test the code without preprocessing. In our example, a compiler would identify unreachable code in Line 20, whereas the lines 18 and 20 are never included in the same product. The brute-force strategy is to run the preprocessor with every possible configuration followed by the actual analyses (e.g., compilation and testing). This strategy would identify the same compiler error in two products and, thus, involve redundant effort. While the brute-force strategy is feasible in our tiny example, it cannot be applied to Linux with up-to $2^{18,000}$ product variants.

In the past two decades, numerous approaches have been proposed to analyze product lines more efficiently than with the brute-force strategy [47]. However, they typically only focus on efficient analysis of variants and not revisions. In contrast, regression analyses focus on revisions, but not variants. In the following, we discuss how to efficiently analyze variation according to a single dimension of variation and with respect to both dimensions of variation.

3 APPLYING PRODUCT-LINE ANALYSES TO VARIATION IN TIME

In this section, we discuss work on product-line analysis, that is, analyses of software systems that explicitly consider variability in space as depicted in Figure 2. Such an analysis exploits knowledge about variability in some way to enable the efficient analysis of product lines as we discuss in Section 3.1.

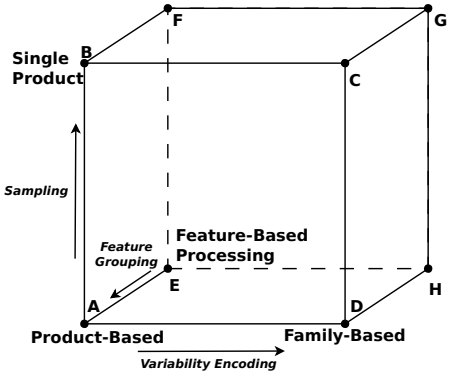

Figure 6: The product-line analysis cube visualizes the space of possible combinations of product-line analyses [53].

In case one wants to analyze different revisions over time in an integrated way, one can regard these different revisions as analogous to variants. This opens the possibility of *applying* product-line analyses to revisions, depicted as an arrow leading from product-line analyses (top left) to regression analyses (bottom right) in Figure 2. We explore this concept further in Section 3.2.

3.1 Product-Line Analyses

The idea of product-line analyses is to establish properties for all products of a product line. Basically any of existing product analysis can be lifted to the product-line level. These can be static analyses like dead-code analyses, which aim at identifying code parts that are not part of any variant [46] or type checking of product lines that aims at identifying whether any variant violates typing rules [11, 27]. For instance, type checking can find the compilation error for configuration c_1 and c_3 discussed in Section 2.

Many different analyses have been lifted to the product line level [47]. However, the basic strategies applied can be divided into three different categories: product-based, feature-based, and family-based analysis as well as combinations thereof [47]. A *product-based analysis* aims at analyzing the property for each product, individually, concluding the property when all products have been shown to exhibit the property. *Feature-based analysis*, in a similar way, analyses the property for each feature in the product line. Finally, *family-based analysis* aims to perform the analysis for the whole range of domain assets in an integrated way, taking the relevant variability explicitly into account.

The product-line analysis cube shown in Figure 6 builds on those basic strategies and extends it to a formalization of product-line analysis strategies [53]. It introduces three different strategies for variability analysis: sampling, feature grouping, and variability encoding. *Sampling* refers to the strategy of selecting a subset of products and performing a product-level analysis on them. Using an adequate sampling heuristic, this aims to establish a high probability that if the property under analysis holds for the subset, it will hold for all products. For instance, it would be sufficient to have either c_1 or c_3 in the sample to detect the type error of our running example. *Feature grouping* refers to the concept of focusing on adequately selected subsets of features. Feature grouping does not necessarily say anything regarding the completeness of the features to be addressed. It can be combined with sampling, that is, only addressing some features in total and making a heuristic argument regarding the completeness, or it can select a subset of features that allow for the sound conclusion that the property holds for all features, if it can be shown for the subset [43]. It can also be a way to subdivide the range of features, but all will be taken into account over the complete analysis. Finally, *variability encoding* describes the strategy of explicitly encoding the variability and using this information within the analysis. The product-line analysis cube allows designers of product-line analyses to choose any point in the cube and to build such an analysis, whereas it depends on the product line and type of analysis which point in the cube leads to the most efficient analysis [53].

3.2 Reusing Product-Line Analyses for Revisions

In principle, we can interpret a set of revisions as a set of products in the sense of a product line. Thus, we can interpret variation in time as variation in space, providing a pathway to apply product-line analysis also to revisions. This way, product-line research and tools may be applied as-is for the analysis of variation in time.

We can regard the differences that are created by updates from one revision to another as features that exist in the product line. For instance, the two revisions of the *MultiStore*, presented in (1) and (2) in section 2, could be threated as two separate features—one that reads only one value from the store and another that can read all values. This would allow for the application of feature-based analysis approaches to revisions. To the best of our knowledge, this has not yet been done, but it would seem to be straightforward with delta-based approaches like delta-oriented product testing [3].

A different angle would be to use a sufficiently expressive variation representation, which allows encoding revisions over time in the same way that we represent variation in space. For example, with an annotative variation representation such as #ifdefs (see Listing 4) or the choice calculus [18], we can introduce a "feature" corresponding to each revision of the program. Similarly, in delta-modeling, revisions can be captured by deltas [31]. By encoding revisions as variants, we can directly apply family-based analyses designed for software product lines to entire revision histories. The ability to efficiently analyze a large number of revisions in time as well as in space opens up new potential applications, such as determining when a particular behavior or interaction was introduced, supporting sophisticated ways of selectively undoing revisions or cherry-picking patches, and more.

However, product-line analyses have so far hardly been applied to the problem of analyzing revisions at scale. Reasons may be that revision histories can be long and family-based analyses are more expensive than the analysis of individual products. Since we usually only care about a limited number of revisions in the history (e.g., major releases or specific time slices), more general approaches to incrementalize and/or constrain family-based analyses are needed to realize the full potential of such approaches. This opens a new field for further research.

4 APPLYING REGRESSION ANALYSES TO VARIATION IN SPACE

In this section, we discuss work related to analyses applied in the context of variability in time, namely *revisions*, as illustrated in Figure 2. We use the term regression analyses to denote works that apply verification and validation techniques over revisions, such as regression testing [54], incremental program analysis [7, 45], and change impact analysis [9, 24]. When considering revisions, the main goal of these techniques is to ensure that the existing software still behaves as expected after a change. We discuss existing approaches which are unaware of variability in Section 4.1. We then present some works that apply regression analyses to the variability context in Section 4.2. This corresponds to the arrow from regression analyses (bottom right) to product-line analyses (top left) in Figure 2.

4.1 Regression Analyses

In theory, to apply regression testing, an approach would be to execute the entire test suite of a system again, to check behavior preservation. However, the number of tests might grow together with a system, making such an approach unfeasible in practice. Therefore, such techniques usually consider what has been changed in a particular evolution scenario, to increase efficiency by defining which tests should be executed, identifying redundant tests, and establishing the priority for ordering test case execution [54]. In our running example, prioritizing a test case containing a call to the *readAll()* function can lead to discovering the fault in the implementation of the function.

A number of works have been proposed targeting incremental analysis, such as separate compilation [13], to avoid recompiling an entire program when there is some change in an interface. Recent works also target incremental program analysis [7, 45], proposing ways for tackling program changes in an efficient way. For instance, *Reviser* [7] extends an existing framework for data-flow analysis to enable recomputing analysis information after a program changes. Its basic intuition is to identify changes based on the control-flow graph of two program versions, and by establishing the origin of changes, proceed on propagating and updating the results of the analysis. *IncA* is a domain-specific language for specifying incremental program analysis, which works in a declarative way over AST representations of programs [45]. For both works, a speedup is observed when compared with running the entire analysis again, while no substantial overhead is introduced. Nonetheless, variability is not considered.

4.2 Reusing Regression Analyses for Variants

If we want to apply regression analyses in the context of variants, one option is to consider variants as revisions, then proceed to apply standard regression analysis. The first issue that arises is the number of variants, which might be challenging to deal with since there might be a huge configuration space. One option is to focus on a subset of the variants to reduce the effort of the analysis. In some cases, it is feasible to consider only those products of interest to particular customers—a common industrial practice [36], and in other cases dedicated sampling techniques are required to derive a sample with certain coverage guarantees [2, 52].

Lity et al. apply regression testing techniques to verify that changes between variants are intended, to systematically exploit reusability of test artifacts [32]. They use delta-oriented models to capture commonalities and variabilities among variants. This approach to modeling is used for reasoning about changes between variants and test artifacts. Testing effort is reduced while maintaining the same degree of test coverage. Heider et al. propose Variability Modeling Regression Testing (VaMoRT) [24]. The approach relies on decision models as variability models, and allows identifying the impact of changes to the variability models on existing products. It avoids testing all variants but rather focuses on the already derived products, since the decision model stores configuration decisions. Based on that, the approach regenerates the products and computes the differences.

Another issue is the actual order in which variants are analyzed. When we are considering variants, change is inherent among them. Therefore, it could be the case that some changes are analyzed more than necessary. Hence, an adequate order in which variants are analyzed may result in minimal changes and avoid rework. There is some work on prioritizing product orders for testing [3, 30]. Al-Hajjaji et al. [3] focus on obtaining higher coverage with the goal of early fault detection. Therefore, it privileges selecting dissimilar products when establishing an order. In contrast, the work by Lity et al. [30] focuses specifically on leveraging an order that might benefit incremental analysis. Thus, it prefers an order in which differences between adjacent products are minimized. Changes among products are captured in regression deltas, and this enables graph algorithms to find an optimal product order. However, this work mainly focuses on comparing products from the same product-line revision. If we extend it to consider revisions of variants, depending on the regression analysis and whether it involves hardware to be reconfigured it could be necessary to have a linear order of variants to be analyzed. This is opposed to a tree-like regression analysis, in which the best match is found for each variant in each of the revisions.

5 EFFICIENT ANALYSES FOR VARIATION IN TIME AND SPACE

In the previous two sections, we discussed how to efficiently analyze the variability induced by either variants or revisions. However, software systems that exist in variants, evolve as well. Thus, as illustrated by the motivating example in Section 2, there is potential and a need for incremental analyses along both dimensions of variation (i.e., time *and* space) simultaneously.

In this section, we aim to discuss different ways in which product-line analyses and regression analyses can be applied to both variants and revisions at the same time. In total, we discuss three different strategies in the following three subsections: in Section 5.1 we discuss how product-line analyses can be homogeneously applied to revisions of variants. In Section 5.2, we complement this by a discussion of how regression analyses can be simultaneously applied to variants and revisions of variants. Finally, in Section 5.3, we discuss the combination of product-line analyses with regression analyses. This structure is also illustrated in Figure 2.

5.1 Applying Product-Line Analyses to Revisions of Variants

While there has been a considerable amount of work on product-line analyses [47], most of it does not aim to save analysis effort when applying it to multiple revisions of a product line. The naïve way of applying any product-line analysis to multiple revisions is to simply analyze every revision from scratch. However, this does not take advantage of typically rather local and comparably small changes to a product line [28]. For instance, in our motivating example, a very small change was made to the implementation of the *MultiStore* feature, affecting only two of the four possible products of the product line; thus, an analysis of the whole product line is unnecessary. Nevertheless, this naïve strategy may serve as a baseline when evaluating the efficiency of more advanced analyses.

One strategy to avoid redundant effort under evolution is to identify which products are affected by the change. A common solution is to classify changes to the product line into refactorings, specializations, generalizations, and arbitrary edits [5, 10, 15, 48]. When applying a refactoring to a product line (i.e., the set of derivable products is identical in both revisions) [49], there is no need to analyze the product line again, as long as it is verified that the change is indeed a refactoring. When applying a specialization (i.e., some products are removed but no new products are added) [15], then an analysis is only necessary if the previous revision did produce errors, as those may be fixed by means of the specialization. For generalizations and arbitrary edits (i.e., changes that add new products), product-line analyses need to run again usually. Borba et al. generalized refactorings, generalizations, and specializations with safe evolution [8], which also incorporate changes to implementation artifacts and not only feature models. Schulze et al. and Pietsch et al. propose a catalog of refactorings [44] and a refactoring construction kit [39, 40] for delta-oriented product line implementations. Sampaio et al.'s extension to partially safe evolution even allows to classify which products are affected by a change [42]. Tool support for classifying changes of the evolution history has been proposed by Dintzner et al. [16].

As product lines can be arbitrarily complex and changes often only have local effects, it has been proposed to modularize product lines into multiple product lines [26, 43, 51]. The overall idea is that a tool checks whether a change is local to an individual product line; if it is indeed local, then it is sufficient to analyze changes only locally and avoid the analysis of all product lines. This imposes some constraints on how the product line is implemented and modeled, but may significantly reduce the effort when analyzing new revisions of the product lines.

Product-line analyses can be organized according to whether their basic unit of analysis is a single product, a feature implementation (e.g., a component or plug-in), or the entire family of products (see Section 3.1). Of these, perhaps *feature-based analyses* best support product-line evolution, because they are inherently incremental. After a revision, one needs only to re-analyze the features that have changed. One challenge is that feature interactions cannot be detected by a feature-based analysis, which is why it is typically combined with a product-based or family-based analysis [47]. So changing a feature implementation may also require re-analyzing other unchanged features that potentially interact with the changed feature. There is likely potential for reuse in these analyses since presumably most code in other features does not interact with the changed feature.

5.2 Applying Regression Analyses to Revisions of Variants

Regression analyses are typically only applied *either* to revisions [7, 9, 24, 45, 54] *or* to variants [4, 47]. Applying a regression analysis to both dimensions of variation is feasible, but there seem to be many opportunities to make it more efficient that are not yet well researched and discussed in the following.

A trivial way to apply regression analysis to revisions of a product line is to follow the strategy described in Section 4.2 to apply it to all variants of a revision and then to apply it from scratch to the next revision, to which we refer to as *variant-only regression strategy*. However, in this case we do not exploit the similarities among the product-line revisions. In contrast, we may exploit the similarities among the product-line revisions, by applying the regression analysis to each variant with respect to its prior revision, to which we refer to as *revision-only regression strategy*. However, then we cannot exploit the similarities among variants within one revision of the product line. Furthermore, it is necessary to have a mapping from variants in the old product-line revision to the variants in the new product-line revision [37]. Such a mapping may not be easy to find depending on the kind of evolution that happened and the implementation technique for variation in space. For delta-oriented product lines, the revision-only regression strategy has been applied in the context of model-based testing [34]. A more advanced strategy, as mentioned in Section 4.2, would be to find the variant with the smallest number of changes to all those that have been analyzed previously. This could include not only analyzed variants of the old product-line revision, but also already analyzed variants from the new product-line version.

As discussed in Section 4.2, the application of regression analyses to product lines typically requires to sample products. When the product line evolves, we can sample the product line again and then apply regression analyses to those sample products. The creation of a new sample can be avoided if none of the artifacts being used as input to the sampling is affected [38]. For instance, if only the feature model is used as input to the sampling algorithm, then we only need to apply the sampling algorithm again on changes to the feature model. However, there are many cases when the computation of a new sample cannot be avoided. While there are many sampling algorithms for product lines [2, 52], they are typically oblivious to the evolution of the product line [38]. In the context of regression analysis, it would be favorable to retrieve a sample for a new revision of the product line that is largely similar to the old sample [37]. The reason is that the efficiency of regression analyses typically depends on the number of changes and similar samples could reduce the number of changes. Besides that, it could be more efficient to adapt the old sample than to start over the sampling algorithm from scratch [37].

5.3 Combinations of Product-Line Analyses and Regression Analyses

So far, we discussed how existing product-line analyses and regression analyses can be reused or extended to support variation in time and space. Besides that, we could combine existing product-line analyses and regression analyses or even invent new product-line regression analyses. Both regression analyses and product-line analyses have in common that they lift an existing analysis to some form of variation, whether variation in time or variation in space. Ideally, we would accomplish an efficient *two-dimensional lifting* by lifting an existing analysis first to variation in time and then to variation in space, or vice versa. Midtgaard et al. proposed to automatically lift existing analyses for variation in space [35], which could be a starting point for the automatic derivation of product-line regression analyses. Two-dimensional lifting has great potential, but it is unclear how much automation is feasible and how efficient automatically derived analyses are.

A direct approach to perform such lifting would be to directly combine an existing product-line analysis with a regression component. As the analysis of real-world product lines shows that individual commits in an evolution history typically have only a very limited impact on the variability in a product line [28], one can expect regression-based extensions of product-line analysis to be very efficient as the updating of analysis results may be a rather local affair. This idea has been applied to dead code analysis over a significant part of the Linux kernel evolution history [20], leading to a speedup over the baseline by an order of magnitude despite the fact that any variability-relevant change of the build model and variability model lead to a complete reanalysis.

Another perspective on product-line regression analyses is to consider time as a fourth dimension in the product-line analysis cube [53]. As discussed in Section 3.1 and illustrated in Figure 6, the existing cube focuses on the mix and match of different analysis strategies with the goal to provide efficient analyses. However, all three existing dimensions, namely variability encoding, sampling, and feature grouping, are mainly focused on variation in space. It is an open research question how to extend the product-line analysis cube to variation in time, but time could be considered as a fourth dimension, or there could be several dimensions for variation in time as there are also three for variation in space.

Another open research question is to what extent the realization techniques for variation in space and variation in time have an impact on the efficiency of product-line regression analyses. For instance, are plug-ins more amendable to analysis than preprocessor annotations? Do we need the change operations that have been applied or is a diff equally good for analysis? Furthermore, it is an open question whether analysis efficiency can be improved if variation in space and variation in time are expressed by the same means, as with higher-order deltas [31] or 175% modeling [33].

6 CONCLUSION AND FUTURE WORK

Today's software development needs to cope with variation in time, variation in space, or even in time *and* space. Crucial for quality assurance and efficient analysis methods that take advantage of commonalities arising from both dimensions of variation. Traditionally, regression analyses are used for variation in time and product-line analyses are used for variation in space.

We discussed fundamental strategies to apply regression analyses and product-line analyses to both dimensions of variation. In particular, we identified how product-line analyses can be applied to analyze variation in time, which seems to be a new application area for many existing product-line analyses. That is, research results of the product-line community could be reused by communities working on regression analyses. While regression analyses have often been applied to variation in space, we summarized common challenges for their application to variants.

With product-line regression analysis, we denote analyses that cope with variation in both dimension, namely time and space. For that purpose, two-dimensional lifting of traditional analyses is necessary with respect to both dimensions of variation. It requires a community effort to identify how to automate the lifting according to both dimensions and which strategies for lifting lead to the most efficient analyses, perhaps also paving the way for an integration of multiple strategies.

ACKNOWLEDGMENTS

In this paper, we summarize the insights of our discussions during a breakout group at Dagstuhl 19191 on *Software Evolution in Time and Space: Unifying Version and Variability Management*. We gratefully acknowledge discussions on sampling for product-line evolution with Sascha Lity, Tobias Pett, Malte Lochau, Sebastian Krieter, and Tobias Runge. This work has been partially supported by the German Research Foundation within the project VariantSync (TH 2387/1-1 and KE 2267/1-1), Science Foundation Ireland (13/RC/2094), FACEPE (APQ-0570-1.03/14), CNPq (409335/2016-9), and the ITEA3-project REVaMP^2, funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H.

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages.

[2] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730.

[3] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *Proc. Int'l Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 34–40.

[4] Nauman bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. 2019. On the Search for Industry-Relevant Regression Testing Research. *Empirical Software Engineering* (12 Feb 2019). https://doi.org/10.1007/s10664-018-9670-1

[5] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. 2006. Refactoring Product Lines. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 201–210.

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[7] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[8] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A Theory of Software Product Line Refinement. *Theoretical Computer Science* 455, 0 (2012), 2–30.

[9] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, Leopoldo Teixeira, and Sabrina Souto. 2018. A Change-Aware Per-File Analysis

to Compile Configurable Systems with #ifdefs. *Computer Languages, Systems & Structures* 54 (2018), 427–450. https://doi.org/10.1016/j.cl.2018.01.002

[10] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2015. Reasoning about Product-Line Evolution Using Complex Feature Model Differences. *Automated Software Engineering* 23, 4 (2015), 687–733.

[11] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Programming Languages and Systems (TOPLAS)* 36, 1, Article 1 (2014), 1:1–1:54 pages.

[12] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *Comput. Surveys* 30, 2 (1998), 232–282.

[13] Regis Crelier. 1994. *Separate Compilation and Module Extension.* Ph.D. Dissertation. Institute for Computer Systems, ETH Zurich. ftp://ftp.inf.ethz.ch/doc/diss/th10650.ps.gz

[14] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.

[16] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 2 (2018), 905–952.

[17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 19–28.

[18] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *Trans. Software Engineering and Methodology (TOSEM)* 21, 1, Article 6 (2011), 6:1–6:27 pages.

[19] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* ACM, New York, NY, USA, 349–360. https://doi.org/10.1145/2642937.2642987

[20] Moritz Flöter. 2018. Prototypical Realization and Validation of an Incremental Software Product Line Analysis Approach.

[21] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM, 279–290.

[22] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* ACM, 323–334.

[23] Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proceedings of the 46th Annual Design Automation Conference (DAC '09).* ACM, New York, NY, USA, 466–471. https://doi.org/10.1145/1629911.1630034

[24] Wolfgang Heider, Rick Rabiser, Paul Grünbacher, and Daniela Lettner. 2012. Using Regression Testing to Analyze the Impact of Changes to Variability Models on Products. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 196–205.

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM, 805–824.

[26] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-Aware Module System. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM, 773–792.

[27] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD).* ACM, 25–32.

[28] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18).* ACM, New York, NY, USA, 54–64. https://doi.org/10.1145/3233027.3233032

[29] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM, 81–91.

[30] Sascha Lity, Mustafa Al-Hajjaji, Thomas Thüm, and Ina Schaefer. 2017. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 60–67.

[31] Sascha Lity, Matthias Kowal, and Ina Schaefer. 2016. Higher-Order Delta Modeling for Software Product Line Evolution. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016).* ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/3001867.3001872

[32] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. 2012. Delta-Oriented Model-based SPL Regression Testing. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering (PLEASE '12).* IEEE Press, Piscataway, NJ, USA, 53–56. http://dl.acm.org/citation.cfm?id=2666064.2666078

[33] Sascha Lity, Sophia Nahrendorf, Thomas Thüm, Christoph Seidl, and Ina Schaefer. 2018. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 27–34.

[34] Sascha Lity, Manuel Nieke, Thomas Thüm, and Ina Schaefer. 2019. Retest Test Selection for Product-Line Regression Testing of Variants and Versions of Variants. *J. Systems and Software (JSS)* 147 (2019), 46–63.

[35] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2015. Systematic Derivation of Correct Variability-Aware Program Analyses. *Sci. Comput. Program.* 105, C (July 2015), 145–170. https://doi.org/10.1016/j.scico.2015.04.005

[36] Mukelabai Mukelabai, Damir Nesic, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* ACM, 155–166. https://doi.org/10.1145/3238147.3238201

[37] Tobias Pett. 2018. *Stability of Product Sampling under Product-Line Evolution.* Master's thesis. TU Braunschweig, Germany.

[38] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM. To appear.

[39] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL–A Delta-Based Modeling Framework for Software Product Line Engineering. In *International Conference on Automated Software Engineering.* IEEE, 852–857.

[40] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM. To appear.

[41] Hendrik Post and Carsten Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* IEEE, 347–350.

[42] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2016. Partially Safe Evolution of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 124–133.

[43] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE).* ACM, 667–678.

[44] Sandro Schulze, Oliver Richers, and Ina Schaefer. 2013. Refactoring Delta-Oriented Software Product Lines. In *International Conference on Aspect-Oriented Software Development.* ACM, 73–84.

[45] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).* ACM, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

[46] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys).* ACM, 47–60.

[47] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (2014), 6:1–6:45.

[48] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE).* IEEE, 254–264.

[49] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* IEEE, 191–200.

[50] Thomas Thüm, André van Hoorn, Sven Apel, Johannes Bürdek, Sinem Getir, Robert Heinrich, Reiner Jung, Matthias Kowal, Malte Lochau, Ina Schaefer, and Jürgen Walter. 2019. Performance Analysis Strategies for Software Variants and Versions. In *Design for Future—Managed Software Evolution.* Springer, Berlin, Heidelberg. To appear.

[51] Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. 2016. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 97–104.

[52] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 1–13.

[53] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. 2013. The PLA Model: On the Combination of Product-Line Analyses. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 14:1–14:8.

[54] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability (STVR)* 22, 2 (2012), 67–120.