Predicting Higher Order Structural Feature Interactions in Variable Systems

Stefan Fischer, Lukas Linsbauer, Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
Stefan.Fischer@jku.at, Lukas.Linsbauer@jku.at, Alexander.Egyed@jku.at

Roberto Erick Lopez-Herrejon
Dept. Software Engineering and IT
École de technologie supérieure
Montreal, Canada
Roberto.Lopez@etsmtl.ca

Abstract—**Robust and effective support for the detection and management of software features and their interactions is crucial for many development tasks but has proven to be an elusive goal despite extensive research on the subject. This is especially challenging for variable systems where multiple variants of a system and their features must be collectively considered. Here an important issue is the typically large number of feature interactions that can occur in variable systems. We propose a method that computes, from a set of known source code level interactions of n features, the relevant interactions involving n+1 features. Our method is based on the insight that, if a set of features interact, it is much more likely that these features also interact with additional features, as opposed to completely different features interacting. This key insight enables us to drastically prune the space of potential feature interactions to those that will have a true impact at source code level. This substantial space reduction can be leveraged by analysis techniques that are based on feature interactions (e.g Combinatorial Interaction Testing). Our observation is based on eight variable systems, implemented in Java and C, totaling over nine million LoC, with over seven thousand feature interactions.**

I. Introduction

Variable software stems primarily from the adoption of generator-based techniques (e.g. [1], [2]), full-blown *Software Product Line (SPL)* approaches (e.g. [3]), advanced modularization paradigms (e.g. [4]), or ad hoc reuse practices collectively called *clone-and-own* [5], [6]. Typical variable software systems are formed with a large number of distinct variants, sometimes in the order of myriads [7]. What distinguishes a variant is the set of *features* – increments in program functionality [8] – that each variant provides. The possible variants that can be derived from a variable system can be described in a *variability model* [9].

The effective development of variable software poses unique challenges to software engineers not only because of the large number of variants that should be considered collectively but because all feature interactions must be analyzed, understood, and managed. Broadly speaking, a feature interaction occurs when the behavior of one feature changes depending on the presence or absence of another feature or set of features [10]. The number of features involved in an interaction determines the *order* of the interaction. The more features interact, the higher the order of the interaction. Feature interactions can cause a wide range of problems if not treated properly; for example, in source code they can lead to unexpected executions, faults, or changes in non-functional properties. Research in feature interactions thus has a long standing history (e.g. [11]–[15]) but there are many open and pressing challenges [11]. Salient among them is combinatorial explosion of potential feature interactions. The number of potential interactions grows with the number of features $|F|$ in a system multiplied by the order of feature interactions: $\sum_{n=1}^{|F|}(\binom{|F|}{n} * 2^n)$. One domain that suffers from this combinatorial explosion is the testing of variable systems. Commonly, Combinatorial Interaction Testing (CIT) is used to compute a subset of variants to test, that cover all n-wise feature-combinations [16]–[18]. For example, pairwise testing computes variants that cover all combinations of two features, selected and unselected. This misses out on 3-wise or 4-wise and the ability to detect faults related to interactions among a larger number of features. Previous work has shown the need to deal with these greater than 2-wise interactions [19], [20]. However, computing them becomes infeasible for large variable systems [21]. Beyond testing, there are many other approaches, often based on search-based algorithms, that struggle with the enormous search space of potential feature interactions [22].

In this paper, we investigate a method for predicting higher-order feature interactions in source code from known lower-order interactions. Our approach assumes that we know interactions among n features to subsequently check if there are feature interactions among $n + 1$ features. For example, in the context of testing, our method could be used to predict which interactions should be considered for 3-wise testing based on the results of pairwise testing (i.e. pairwise feature combinations that lead to faults should also be tested in combinations with other features). More generally, our method can be used to extend or complete the results of incomplete analyses (e.g. in cases where a complete analysis would have been infeasible) for finding structural feature interactions. To evaluate our method we employed eight variable systems totaling over nine million LoC, implemented in Java and C, and analyzed over seven thousand feature interactions. Our study revealed a clear connection between lower- and higher-order interactions with many practical implications.

II. Background

In this section we provide the background and basic terminology required for our empirical assessment along with an illustrative running example.

A. Features and Feature Interactions

Within the realm of variable software there are many definitions or interpretations for the concept of features [23]. Because the focus of our study is on source code annotations, we use Zave's definition as our running definition and regard features as increments in program functionality [8]. Hence, for our study, features are source code increments that implement a given program functionality. Similarly, there are also multiple conceptions and interpretations of the concept of feature interactions [11]. According to the terminology proposed by Apel et al. the focus of our study is on *structural interactions* that manifest in source code that is present when features interact [10].

Definition 1: A *structural feature interaction* manifests at source level whenever source code is included in a software variant because of a combination of selected and unselected features of the variant.

Henceforth and unless otherwise stated, whenever we say feature interaction we mean structural feature interaction.

In order to describe and analyze the structure of features and their interactions we introduce the notion of *modules* to label the different relationships that can be identified in the implementing source code. This notation and terminology is further explained in [5].

Definition 2: A *module* $\delta^{\mathrm{n}}(\mathrm{c}_0, \mathrm{c}_1, ..., \mathrm{c}_\mathrm{n}) = \{c_0, c_1, ..., c_n\}$ labels a feature or feature interaction, where c_i is $\mathtt{F}$ (if feature $\mathtt{F}$ is positive, i.e. selected) or $\neg\mathtt{F}$ (if negative, i.e. not selected), and n is the order of the module. A module contains at least one feature.

The order of a module denotes how many features interact in that module. A module of order n represents the interaction of $n + 1$ features. Modules with order 0 label a single feature and not an interaction between features.

B. Variability Models (VMs)

An important part of variable software is to specify the set of different variants that it comprises. That is the role of *variability models*. There are several alternatives (e.g. [7], [9]). One of the most commonly used are *feature models* that are hierarchical tree-like structures where nodes represent features and edges denote different types of variability relations [1]. Variability models can be formally described, for instance using propositional logic, which enables formal reasoning about their properties. For example, counting the number of feature combinations or verifying if a variant with a certain partial feature combination exists. For further details please refer to [24].

C. Running Example

For illustration of these concepts consider our running example, a small SPL of drawing applications based on [5]. All configurations of this SPL consist of different combinations of six features: BASE represents the basic framework that all configurations have, features LINE and RECT enable the drawing of lines and rectangles respectively, feature WIPE allows the user to wipe the drawing area clean, and feature COLOR adds functionality for choosing different colors to draw with. Finally, feature FILL allows the user to select a color to fill a rectangle with.

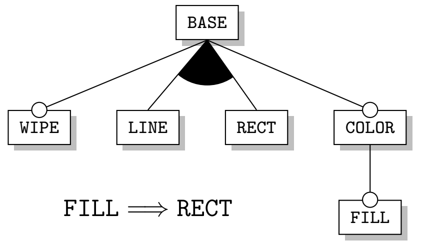


Fig. 1: Feature Model for our Running Example

The variability model of our example is illustrated in Figure 1, as a feature model. In our example, we only use three types of feature relations: *i.* optional features which may or may not be selected if their parent feature is selected, like WIPE, COLOR and FILL, denoted with the empty circle at the top of the feature node, *ii.* inclusive-or groups, where at least one member of a group of features must be selected when their parent is selected, like LINE and RECT, denoted with the filled arc between the feature connections, and *iii.* a cross-tree-constraint that describes the that for feature FILL to be selected also feature RECT has to be selected.

Figure 2 shows a preprocessor annotated code snippet of our running example. The preprocessor annotations (highlighted in blue) mark which code parts are responsible for implementing which features. Note that we neither require nor assume systems under analysis to be annotated with features. Here we use annotations solely for the sake of illustration. For example, the field definition `List<Line> lines` shown in Lines 2-4 will be included in class `Canvas` of *all* the variants that include feature LINE, independent of any other features being present. Hence this field definition is denoted with the module $\delta^0(\mathtt{line})$.

Consider now the code in Lines 13 and 14. For this piece of code to be included in class `Line` of a variant the conditions in Lines 10 and 12 must hold. This means that both features LINE and COLOR must be selected by such variant, hence these two lines of code represent a *structural feature interaction* between the features LINE and COLOR, denoted with the module $\delta^1(\mathtt{line}, \mathtt{color})$. Similarly, if in a variant the feature LINE is selected but the feature COLOR is not, the Lines 16 and 17 will be included in class `Line`. Hence, these two lines are denoted with module $\delta^1(\mathtt{line}, \neg\mathtt{color})$, because they represent feature LINE if feature COLOR is not selected. For a detailed explanation of the rationale behind negative features please refer to [25].

```
class Canvas {
  #ifdef $LINE
  List<Line> lines;   // δ^0(line)
  #end
  #ifdef $RECT
  List<Rect> rects;    // δ^0(rect)
  #end
  ...
}
#ifdef $LINE
class Line {
  #ifdef $COLOR
  // δ^1(line, color)
  Line(Color c, Point start) {...}
  #else
  // δ^1(line, ¬color)
  Line(Point start) {...}
  #end
  ...
}
#end
#ifdef $RECT
class Rect {
  Rect(
    #ifdef $COLOR
    Color c, // δ^1(rect, color)
    #ifdef $FILL
    Color fillColor, // δ^2(rect, color, fill)
    #end
    #end
    int x, int y){...}
  ...
}
#end
```

Fig. 2: Example of preprocessor annotations

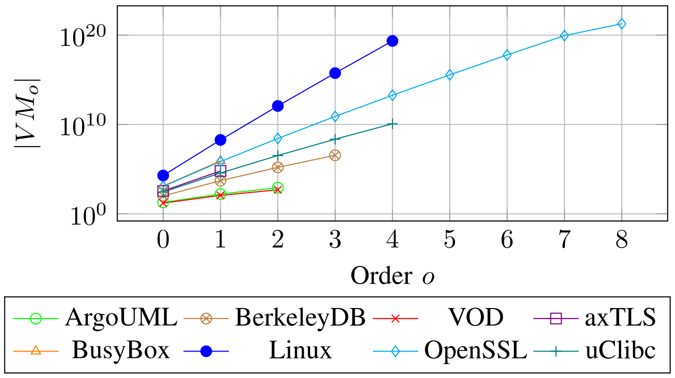


Fig. 3: Number of potential interactions according to the Variability Model

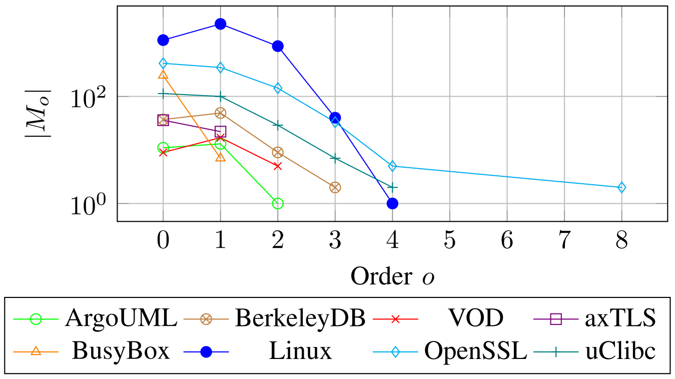


Fig. 4: Number of actual interactions (i.e. modules) found in the source code

III. Problem Description and Research Question

The typically large number of feature interactions render the computation of all of them infeasible. There can exist as many as $\sum_{n=1}^{|F|}(\binom{|F|}{n} * 2^n)$ interactions, were F is the set of all features in the system.

Variability models serve to constrain the ways variable systems can be configured. Therefore, some feature interactions that could potentially exist, might not be allowed according to the variability model. Figure 3 illustrates the number of feature interactions that could exist according to the variability models of the systems in our study. Even though a variability model can reduce the number of possible interactions, the search space is still extremely large, which makes computing all these potential interactions infeasible. For example, in CIT the computation of all pairwise interactions is potentially still possible. However, for large systems, computing all 3-wise interactions might not be possible in reasonable time.

In our previous work we found that only a small fraction of the potential interactions (according to the variability models) is actually present in the source code (see Figure 4) [26]. Our goal in this paper is to find a method to reduce the number of interactions that need to be considered by identifying those interactions (i.e. modules) that likely won't exist in the source code. We do this by using known lower order modules to predict the existence of next higher order modules. In short, our research question is:

RQ. Can higher order modules be directly predicted from lower order modules? *Rationale:* We want to know how well we can predict higher order modules, when knowing the interactions of the next lower order, and therefore potentially reduce the number of interactions that have to be managed. For instance, in our running example, the feature model allows for module $\delta^1(\texttt{line}, \texttt{rect})$, because a variant containing both of these features can be configured. However, there is no structural feature interaction in source code to represent this module and therefore does not need to be considered by any feature interaction analysis (e.g. for CIT).

This prediction can be defined as follows:

Definition 3: Prediction: The set of predicted modules (of the next higher order to o) P_{o+1} is computed as: $P_{o+1} = \{m_{o+1} \mid (m_o \in M_o) \wedge (m_o \subset m_{o+1}) \wedge (|m_o| + 1 = |m_{o+1}|) \wedge (m_{o+1} \in VM_{o+1})\}$. Where M_o is the set of all modules of order o that exist in source code. The variability model constrains the modules that are possible. Therefore, it must also hold that: $M_o \subseteq VM_o \wedge P_{o+1} \subseteq VM_{o+1}$, and VM_o is the set of all modules of order o that can exist according to the variability model.

For instance, in our running example, the module $\delta^1(\texttt{rect}, \texttt{color})$ is present in source code. Using the prediction as defined above we will predict the modules $\delta^2(\texttt{rect}, \texttt{color}, \texttt{fill})$, $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{fill})$, $\delta^2(\texttt{rect}, \texttt{color}, \texttt{wipe})$, $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{wipe})$, $\delta^2(\texttt{rect}, \texttt{color}, \texttt{line})$, $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{line})$, and $\delta^2(\texttt{rect}, \texttt{color}, \texttt{base})$. The module $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{base})$ is not predicted, because it is not valid according to the variability model (i.e. condition $P_{o+1} \subseteq VM_{o+1}$ does not hold).

The rationale here is that, when modules $\delta^1(\texttt{color}, \texttt{fill})$, $\delta^1(\texttt{wipe}, \texttt{fill})$ and $\delta^1(\texttt{wipe}, \texttt{color})$ do not exist in source code (i.e. these features do not interact pairwise), then it is unlikely that module $\delta^2(\texttt{wipe}, \texttt{color}, \texttt{fill})$ (i.e. an interaction between all three features) exists. But when there is $\delta^1(\texttt{color}, \texttt{fill})$ and possibly also $\delta^1(\texttt{wipe}, \texttt{fill})$ and $\delta^1(\texttt{wipe}, \texttt{color})$ in the source code then it is increasingly likely that also $\delta^2(\texttt{wipe}, \texttt{color}, \texttt{fill})$ exists.

IV. Study Setup

This section describes how our empirical study was set up. We present the metrics used for addressing our research question, the variable systems that constitute the corpus of our study and describe the process employed to collect the data. The results and analysis are presented in Sections V and VI respectively.

A. Metrics

Let us now concisely describe the metrics we employed for our study. First we introduce a classification of modules used to compute our metrics. This classification serves to distinguish modules that have been predicted correctly or wrongly, and respectively modules that were not predicted correctly or wrongly. Therefore, when using the modules m_o of order o from source code as the basis for our prediction, the classifications is defined as follows:

- **True Positive (TP):** The set of predicted modules, for which a *structural interaction* exists in source code. Using the terminology of Definition 3:
  $TP_{o+1} = \{m_{o+1} \mid (m_{o+1} \in P_{o+1}) \wedge (m_{o+1} \in M_{o+1})\}$.
  From our running example above, TP_2 only contains module $\delta^2(\texttt{rect}, \texttt{color}, \texttt{fill})$.
- **True Negative (TN):** The set of modules not predicted, that also do not exist in source code, but could exist according to the variability model.
  $TN_{o+1} = \{m_{o+1} \mid (m_{o+1} \notin P_{o+1}) \wedge (m_{o+1} \notin M_{o+1}) \wedge (m_{o+1} \in VM_{o+1})\}$.
  For instance, module $\delta^2(\texttt{wipe}, \texttt{color}, \texttt{fill})$ could exist according to the variability model, but does not exist in source code and was not predicted, because none of the modules $\delta^1(\texttt{color}, \texttt{fill})$, $\delta^1(\texttt{wipe}, \texttt{fill})$, or $\delta^1(\texttt{wipe}, \texttt{color})$ exist.
- **False Positive (FP):** The set of predicted modules, for which no *structural interactions* exists in source code.
  $FP_{o+1} = \{m_{o+1} \mid (m_{o+1} \in P_{o+1}) \wedge (m_{o+1} \notin M_{o+1})\}$.

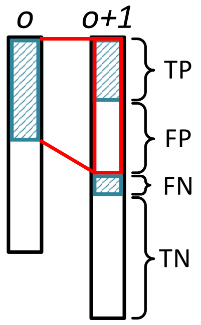


Fig. 5: Visual depiction of interaction prediction

  Therefore, in the example FP_2 contains the modules $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{fill})$, $\delta^2(\texttt{rect}, \texttt{color}, \texttt{wipe})$, $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{wipe})$, $\delta^2(\texttt{rect}, \texttt{color}, \texttt{line})$, $\delta^2(\texttt{rect}, \texttt{color}, \neg\texttt{line})$, and $\delta^2(\texttt{rect}, \texttt{color}, \texttt{base})$.
- **False Negative (FN):** The set of *structural interactions* that exist in source code, but no module has been predicted for them.
  $FN_{o+1} = \{m_{o+1} \mid (m_{o+1} \notin P_{o+1}) \wedge (m_{o+1} \in M_{o+1})\}$
  An example for this would be, if module $\delta^2(\texttt{wipe}, \texttt{color}, \texttt{fill})$ existed in source code, despite none of the models $\delta^1(\texttt{color}, \texttt{fill})$, $\delta^1(\texttt{wipe}, \texttt{fill})$, or $\delta^1(\texttt{wipe}, \texttt{color})$ existing, and therefore it would not be predicted.

Figure 5 depicts the prediction. On the left we illustrate the set of all *interactions* of a given order o, as the outer black rectangle, and the set of *structural interactions* found in code is illustrated as the shaded blue rectangle. The red frame on the right shows the *modules* of order $o+1$ that were predicted. And again, the shaded blue rectangles represent the *structural interactions* we found in source code. Moreover, to further illustrate the different classifications of *interactions* (TP, TN, FP, and FN) we also show them in Figure 5.

Now, based on this classification, we define our metrics as follows:

- **Number of Modules per Order** $|M_o|$. The number of modules of order o identified in the source code of variable systems.
  $$|M_o| : |TP_o| + |FN_o|$$
- **Number of Modules predicted per Order** $|P_o|$. The number of modules of order o that were predicted by our process.
  $$|P_o| : |TP_o| + |FP_o|$$
- **Prevalence** is the proportion of *modules* found in source code with respect to all *modules* possible according to the variability model.
  $$Prevalence_o : \frac{|TP_o|+|FN_o|}{|TP_o|+|TN_o|+|FP_o|+|FN_o|}$$
- **Sensitivity** (or Recall) is the probability that a *module* was predicted, given that it is found in source code.
  $$Sensitivity = \mathrm{P}(predicted \mid present)$$
  OR
  $$Sensitivity_o : \frac{|TP_o|}{|TP_o|+|FN_o|}$$

- **Specificity** is the probability that a *module* is not predicted, given that it is not found in source code.
$$Specificity = \mathrm{P}(notpredicted \mid notpresent)$$
OR
$$Specificity_o : \frac{|TN_o|}{|TN_o|+|FP_o|}$$
- **Positive Predictive Value (PPV)** (or Precision) is the probability that a predicted *module* can be found in source code.
$$PPV = \mathrm{P}(present \mid predicted)$$
OR
$$PPV_o : \frac{|TP_o|}{|TP_o|+|FP_o|}$$
- **Negative Predictive Value** is the probability that a *module* to was not predicted can not be found in source code.
$$NPV = \mathrm{P}(notpresent \mid notpredicted)$$
OR
$$NPV_o : \frac{|TN_o|}{|TN_o|+|FN_o|}$$
- **Accuracy** is the proportion of correctly predicted *modules*, both absent and present.
$$Accuracy_o : \frac{|TP_o|+|TN_o|}{|TP_o|+|TN_o|+|FP_o|+|FN_o|}$$
- **Reduction** is the proportion of predicted *modules*, compared to all *interaction* in the variability model.
$$Reduction_o : \frac{|TP_o|+|FP_o|}{|TP_o|+|TN_o|+|FP_o|+|FN_o|}$$
- **Times predicted**. The number of times a *module* is predicted.

B. Study Corpus

Table I lists all systems that form the corpus of our study, which includes systems implemented in Java and C, two of the most common languages used in variable systems. The used case study systems have their source code annotated with features such that the analysis of structural feature interactions is computationally feasible and hence it can be used as ground truth for the evaluation of the effectiveness of our prediction approach.

The selected systems have been extensively used for different purposes as summarized in the related work (see Section VII). For our selection we had two basic requirements: *i)* publicly available and complete source code, and *ii)* publicly available variability model. To our surprise the selection process proved to be a challenging task because the great majority of existing research either focuses on the extraction of variability models from different sources (e.g. configuration files) or on specialized analysis of variable source code artifacts. This split is in part due to the fact that in most cases the research is carried out by independent groups. A particularly thorny issue was the use of different system versions. This situation resulted in an unexpected difficulty to find variability models that were consistent with the implementation. Just to give an example, a common problem we faced was that the names of the features in the variability model did not match the names of the variables used in the pre-processor annotations. The selected systems for our study were then those that met our two requirements and those for which we were able to reconcile any requirement discrepancies.

Let us briefly describe the selected systems. ArgoUML is an open source project that has been made into a product line of UML modeling tools [29]. BerkeleyDB is a Java implementation of the popular database software library. VOD is a product line for video-on-demand streaming applications. axTLS is an open source library for SSL and TLS protocol support designed especially for platforms with small memory requirements. BusyBox is an open source C library providing an environment for small or embedded systems. Linux is a version of the operating system kernel source code. OpenSSL is an open source project for providing commercial-grade SSL and TLS support as well as a full-strength general purpose cryptography library. uClibc is an open source C library for developing embedded Linux systems. The data of our study corpus is available online[1].

C. Process Overview

In this section we describe the process followed to gather the metrics' data for our study. The process is depicted in Figure 6. The input to our process is the annotated code (1) and a variability model (2) which are handled by the variability aware parser (3) that we developed for extracting all contained structural feature interactions (4) that are used as ground truth for comparison with the results of our proposed method in order to evaluate its effectiveness. Note that when applying our method in practice we do *not* assume that the source code of systems under analysis is annotated. We had to consider a special case for parsing the systems BusyBox, Linux and uClibc. The process for these systems was complicated by the fact that the selection of which files to process was handled by complex make and configuration files based on the selection of features. To address this issue we relied on the information provided by Kästner et al. who tag source code files with *presence conditions* which are propositional formulas that determine when a file is processed or not [30]. We considered the presence conditions when computing the modules based on the annotations.

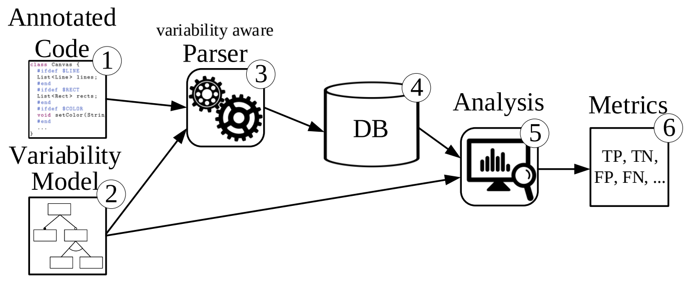


Fig. 6: Process for analyzing SPLs

In the next step, (5), we use extracted structural interactions of one order to predict the modules of the next higher order, and analyze the results in order to compute the defined metrics (6) for each system. The prediction simply works by adding each feature from the variability model, once positive and once

[1]https://github.com/jku-isse/ICSME2018-Predicting-Feature-Interactions

System	Lang	$\|F\|$	$\#LoC$	$\|M\|$	Source
ArgoUML	Java	11	180.4K	25	http://argouml-spl.tigris.org/
BerkeleyDB	Java	55	45K	97	www.oracle.com/technetwork/database/berkeleydb
VOD	Java	11	4.6K	32	Based on [27], [28]
axTLS 1.2.7	C	90	28K	65	http://axtls.sourceforge.net//
BusyBox 1.18.5	C	651	333.3K	395	http://www.busybox.net/
Linux 2.6.33.3	C	11004	8176.3K	5608	https://www.kernel.org/
OpenSSL 1.0.1c	C	589	381.9K	962	https://www.openssl.org/
uClibc 0.9.33.2	C	137	315.9K	259	http://www.uclibc.org/

Lang: Implementation language, $|F|$: Number of Features, $\#LoC$: Number of Lines of Code, $|M|$: Number of modules (i.e. *structural interactions*) found in source code

TABLE I: Variable Systems Overview

negative to each *interaction*, and checking if these predicted modules are valid according to the variability model.

For some systems (OpenSSL, uClibc, and Linux) it was not feasible to compute all theoretically possible *interactions* for each order in a reasonable amount of time due to the large number of features and the complexity of the constraints in their variability models [7]. For these systems we used an estimation similar to the one used by Henard et al. [21] and in our previous work [26]. For this estimation we computed 1000 random *interactions* for each order o and checked how many of these random modules are valid in the variability model. The number of *interactions* can be directly estimated based on the total number of possible *interactions* which is at most $\binom{|F|}{o+1} * 2^{o+1}$. That is, the number of combinations of $o+1$ features (since the order is the number of features minus one) in all $|F|$ features, times the possible combinations of features (negative and positive). For instance, if 800 of the 1000 samples exist in the variability model, we estimate the number of *interactions* of order o to $\frac{800}{1000} * \binom{|F|}{o+1} * 2^{o+1}$. We utilized this estimation whenever the number of possible *interactions* ($\binom{|F|}{o+1} * 2^{o+1}$) is greater than three times the sample size (i.e. 3000).

In this regard we should remark a few points. For computing the lines of code ($\#LoC$) we used the *Count Lines of Code* tool [31], which distinguishes between blank lines, comments and actual code. In Table I, we reported the number of actual lines of code. To check the variability models we used the SAT solver *Sat4J* [32].

V. Results

In this section we present the results we obtained for the metrics introduced above.

We begin by presenting the prevalence of *structural feature interactions* existing in source code, in relation to all potential *interactions* according to the variability model. Figure 7 shows that the prevalence is highest for modules of order 0 (i.e. implementation of single features). For the subsequent higher orders the prevalence decreases significantly. This is due to the increasing number of possible potential *interactions* for higher orders according to the variability model (see Figure 3), compared to the number of modules actually found in source code (see Figure 4).

Figure 8 shows the number of *modules* that have been predicted with our process. When comparing these numbers with the number of potential interactions according to the

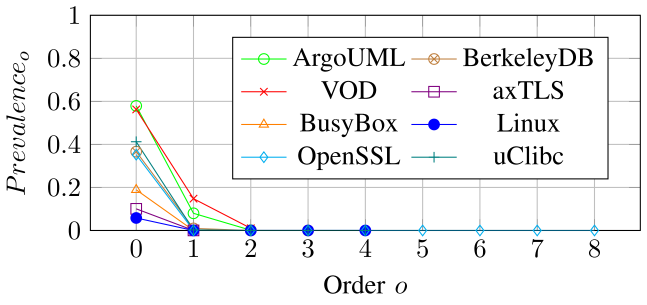


Fig. 7: Prevalence

variability model, in Figure 3, we see a clear reduction of the search space of interactions to consider. Moreover, the number of modules does not exponentially increase with their order like in Figure 3, but appears rather constant and is actually decreasing for higher orders, because fewer modules of higher order are present in source code as basis for prediction (see Figure 4).

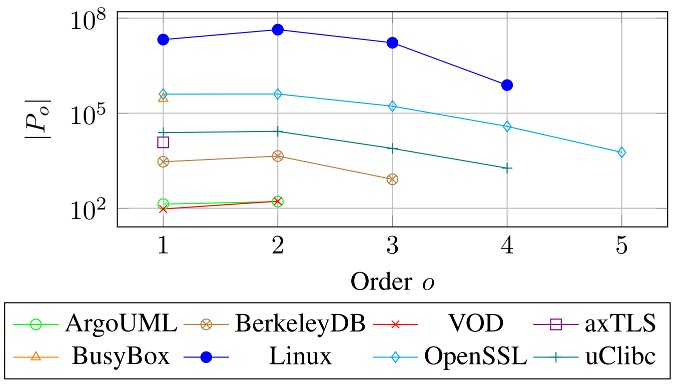


Fig. 8: Number of modules predicted per order

In Figure 9 we illustrate the results we obtained for the rest of our metrics. For Sensitivity in Figure 9a we can see generally good results. This tells us that our prediction includes the vast majority of the next higher order *structural interactions* that are also present in the source code.

Next the Specificity in Figure 9b shows good results for higher order modules. This shows that for modules that do not occur in source code, our prediction has a high probability of excluding them. For modules of order one we see much lower values. This is because nearly all features, from the variability model, are implemented in source code. In combination with the falling Prevalence for *structural interactions* in source code (see Figure 7), this leads to a higher number of modules in FP. Therefore, our prediction will include many modules that are not found in source code, compared to ones that were correctly excluded.

For the Positive Predictive Value (PPV) we observed very low values. In Figure 9c we see that already starting at order one the values are low, and only decrease for higher orders, swiftly converging to zero. However, we expected these results because our prediction will include all modules of the next higher order by adding every feature to the current order modules. Because of the higher Prevalence of lower order modules the PPV is slightly higher for lower orders. Nevertheless, the prediction will oversample the next order modules.

On the other hand, the Negative Predictive Value (NPV) is very high, as shown in Figure 9d. This is because our prediction correctly excludes a high proportion of modules (i.e. large number of modules in TN).

The Accuracy is shown in Figure 9e. Just like the Specificity it gets worse for order one, because of an over prediction and therefore a higher number of false positives. However, the following higher orders generally have high Accuracy, mostly because our prediction is able to exclude a high proportion of potential *interactions* (i.e. large number of modules in TN).

Figure 9f shows the Reduction metric. For each order and system, it shows the proportion of the search space according to the variability model that still needs to be considered according to our prediction. We can see that, especially for higher orders, only a fraction of the search space has to be considered, because our prediction was able to exclude large numbers of interactions.

Next we investigated if we can further improve these results. Especially the results for the PPV are extremely low, because of a high number of wrongly predicted modules (i.e. large number of modules in FP). Therefore we looked for ways to reduce the number of modules in FP. As a simple heuristic for this reduction, we looked into the number of times modules have been predicted and compared these numbers for modules predicted correctly (i.e. in TP) against the numbers for modules predicted wrongly (i.e. in FP). In Figure 10 we present the numbers of how often modules in TP and FP have been predicted, in form of a combination of a violin plot and a box plot. We observed that generally the modules in TP have been predicted more often on average than the modules in FP.

To test this observation we apply the Wilcoxon-Rank-sum test [33]. This confirmed that there is a statistically significant difference in the number of times modules in TP and FP have been predicted. The modules in TP have been predicted more often than the modules in FP. In order to properly interpret the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we used the non-parametric effect size measure $\hat{A}_{12}$ statistic proposed by Vargha and Delaney [34].

System	$\hat{A}_{12}(TP, FP)$
ArgoUML	0.8828874
BerkeleyDB	0.8753485
VOD	0.8217803
axTLS 1.2.7	0.8159237
BusyBox 1.18.5	0.947931
Linux 2.6.33.3	0.5636857
OpenSSL 1.0.1c	0.7113716
uClibc 0.9.33.2	0.7869098

TABLE II: Effect size measures, comparing numbers modules predicted in TP vs. FP

This $\hat{A}_{12}$ value measures the probability that the results for a metric M are higher for one algorithm A versus another B, as a value between zero and one. For instance, $\hat{A}_{12} = 0.3$ means that algorithm A results in higher values for metric M, 30% of the times. If the results for M of the two algorithms are equal, then $\hat{A}_{12} = 0.5$. In general, if $\hat{A}_{12} < 0.5$ $M(A)$ is worse, if $\hat{A}_{12} > 0.5$ $M(B)$ is worse.

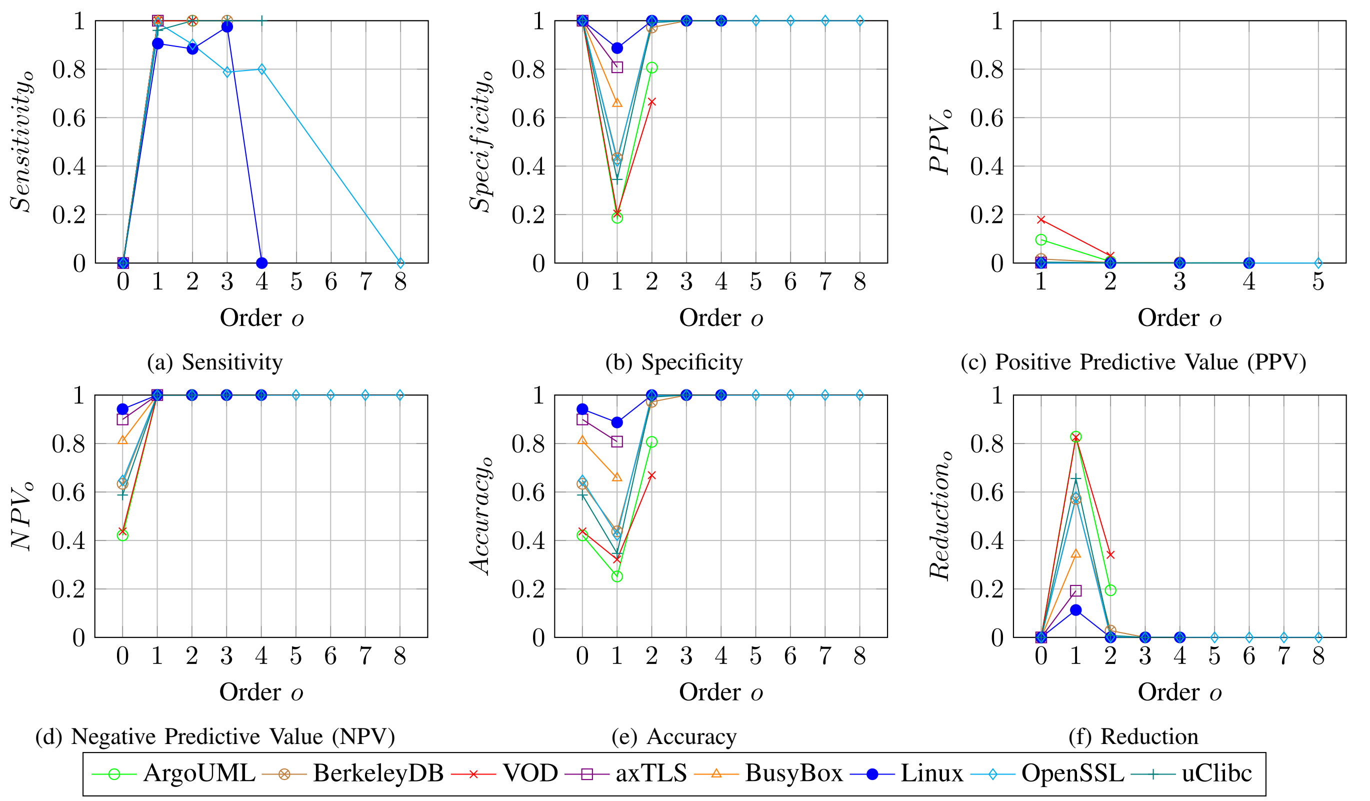
The results for the $\hat{A}_{12}$ measure can be seen in Table II. This shows us that for all systems the modules in TP have been predicted more often than the modules in FP. However, for Linux the $\hat{A}_{12}$ measure is very close to 0.5, which tells us that the difference is not very big. Similarly, in Figure 10, the plots for TP and FP appear very similar for Linux.

Next we introduced a threshold to only include modules that have been predicted at least two times, into our results. We use two as a threshold, because for most systems the majority of modules in TP have been predicted two times. This means we will exclude all modules that have been predicted only once, from our prediction results. Figure 11 shows the number of modules that have been predicted when using this threshold. As expected, the number of modules has been further reduced, when comparing to the predicted modules without the threshold in Figure 8.

In Figure 12 we illustrate the metrics when applying this threshold. We can see that the Sensitivity decreases when filtering the modules in this way in Figure 12a (compared to Figure 9a). This is because the prediction will exclude all the modules that have been in TP before, that have only been predicted once. So these modules moved from TP to FN.

However, the Specificity, the PPV, and the Accuracy are increased, because we were able to correctly reduce the size of FP, by excluding the modules that have only been predicted once. These modules moved from FP to TN. The NPV only changes insignificantly because TN is so much larger than FN, with and without the threshold, and therefore the threshold has very little impact on this metric. Moreover, the Reduction is decreased, which means we reduced the relevant search space even further.

According to the Wilcoxon Signed Rank test the difference of the metrics with and without the threshold are statistically significant [33]. However, the Sensitivity decreases when we introduce the threshold, in some cases the decrease is dramatic, like uClibc with an average decrease of 68%. On the other hand for the systems ArgoUML and BusyBox there is no



(a) Sensitivity (b) Specificity (c) Positive Predictive Value (PPV)

(d) Negative Predictive Value (NPV) (e) Accuracy (f) Reduction

Fig. 9: Interaction Metrics per Order

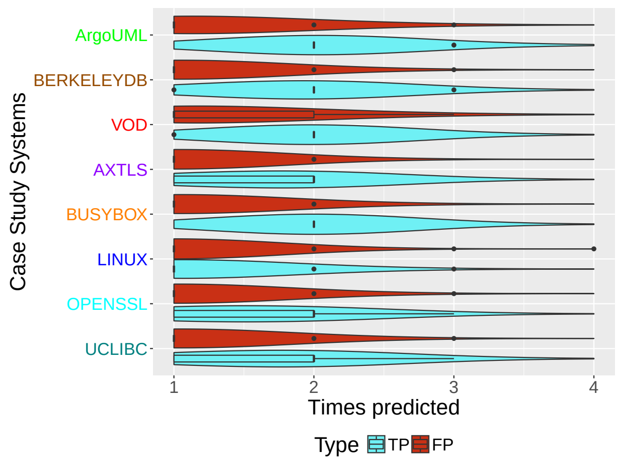


Fig. 10: Number of times modules have been predicted

decrease in Sensitivity, and a strong increase in Specificity and Accuracy, as well as an even stronger Reduction of the search space. For these two systems introducing the threshold into our prediction was a clear gain. However, for other systems it is not so clear and for some the prediction does definitively appear to work better without a threshold.

VI. Discussion

In this section we discuss the findings of our study. We go into implications and limitations, as well as threats to validity.

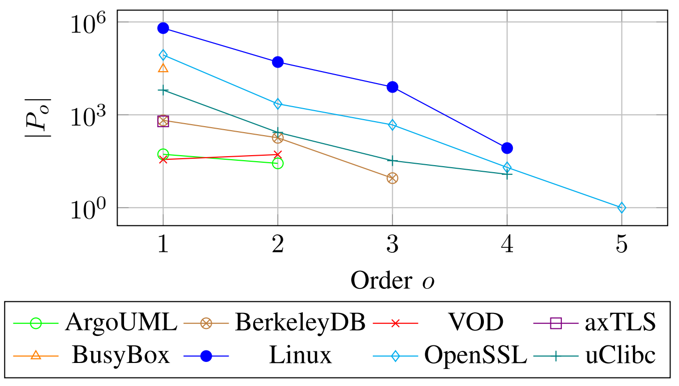
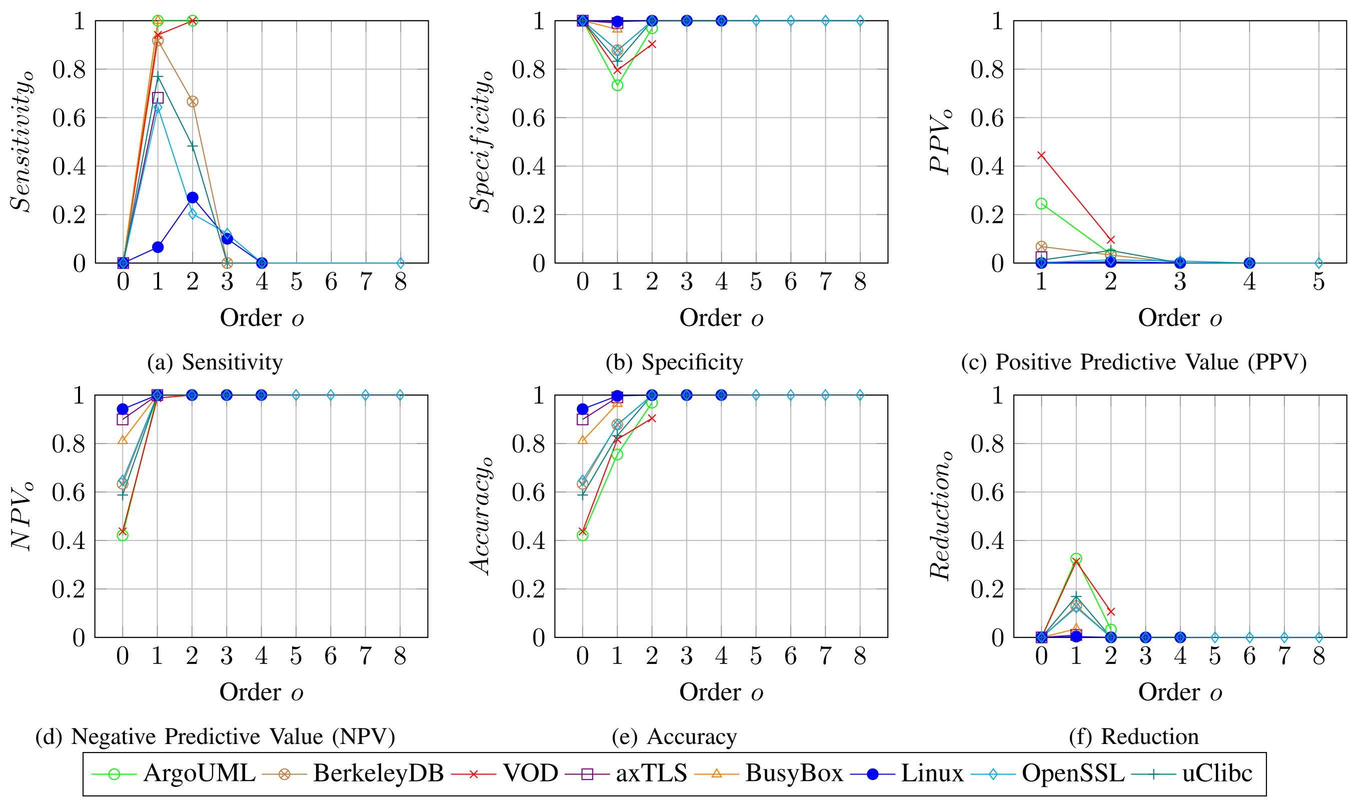


Fig. 11: Number of modules predicted at least twice per order

A. *Implications of findings*

The high number of interactions that we were able to exclude correctly (i.e. high NPV) implies that our prediction excludes a large number of interactions that would have to be considered when working with these SPLs, but do not exist in their source code. We demonstrated that, even though our prediction results in an over sampling of modules (i.e. low PPV), still only a small fraction of the potential feature interaction of the variability model have to considered. Moreover, we found that our prediction eliminates the problem of combinatorial explosion in the number of potential feature interactions, as can be seen in Figure 8. The number of interactions with our



(a) Sensitivity (b) Specificity (c) Positive Predictive Value (PPV)

(d) Negative Predictive Value (NPV) (e) Accuracy (f) Reduction

Fig. 12: Interaction Metrics per Order, when only using interactions that can be predicted at least twice

prediction stays rather constant, no matter how many features are involved. These findings can help to reduce the effort for working with variable systems significantly and make the consideration of higher order modules computationally more feasible. We believe that our work opens many avenues for future research in the area of variable software systems.

The introduction of a threshold into the prediction further improved our metrics, because it reduced the number of modules that were falsely predicted (i.e. previously in FP). The exception to this is the Sensitivity, because also modules previously in TP were rejected for prediction. This means there is a trade-off that has to be decided on a case by case basis. For instance for Linux, the Sensitivity drops hard when introducing the threshold. On the other hand, for the systems ArgoUML and BusyBox, the Sensitivity did not change at all, because all modules in TP were already predicted at least twice without a threshold, as can be seen in Figure 10.

B. Limitations

One limitation to keep in mind with our study is that we only focused on structural feature interactions. There exist other types of feature interactions that are not covered by our study.

Another limitation is, that it is necessary to know modules of a specific order that exist in source code in order to make predictions for the next higher order. However, cases where we know lower order modules but not higher order ones might not be very common. For instance, the static analysis of source code annotations that we used in our evaluation is computationally feasible and identifies all structural interactions of all orders. Nevertheless, we believe that our findings can be very useful for practitioners working with variable systems, in order to reduce the number of interactions to consider and focus their efforts more on actual interactions in source code. For instance, software engineers could use their expertise in a particular domain to specify known feature interactions, and then use a prediction to identify interactions of the next higher order that are more likely to exist. In the context of interaction testing one could first perform pairwise feature tests and then only test higher order interactions based on the results, i.e. only test higher order interactions for which the corresponding lower order interactions already caused tests to fail, or at least prioritize tests of those interactions that have already failed at lower orders. In general, any kind of analysis that can be performed only up to a certain order before becoming computationally infeasible can benefit from our proposed method. Therefore, a myriad of feature interactions that can potentially exist can be excluded and the focus can be put on interactions that are more likely to actually exist.

C. Threats to Validity

In this subsection we describe the threats to validity that we identified in our work, based on the guidelines presented in [35], and how we addressed them.

The first threat was the selection of the variable systems considered in our study. We addressed this threat by including systems that have been extensively studied, see Section VII, and hence are deemed representative of variable software. We included all systems for which complete source code and consistent variability models were publicly available and those for which we found ways to achieve internal consistency, as we described in Subsection IV-C. Certainly, other variable systems might have yielded different results. As part of our future work we plan to expand our study to include more variable systems, also ones implemented in other programming languages.

The second threat can stem from the elaborated process we used to gather and analyze the data as described in Subsection IV-C. We addressed this threat by performing multiple cross-checks and validations both on our data and process as well as those coming from third parties. We are confident that any inconsistencies and inaccuracies were duly identified and appropriately dealt with.

The third threat is the choice of metrics to answer our research questions. For our study, we selected metrics that are widely used, for example, to evaluate the usefulness of medical tests to make a diagnosis [36]. We are confident that these metrics can be used to evaluate the usefulness of our prediction as well. Moreover, we used statistical tests and effect size measures that have been recommended by Arcuri and Briand [37].

VII. Related Work

There is an extensive body of research related to our work. In this section we briefly describe those pieces of work closest to ours categorized by main topics.

Variable Software Analysis. In recent years, there has been an increasing interest in devising strategies to analyze SPLs. A survey and classification for *static strategies* has been proposed by Thüm et al. that considers over a hundred research articles [38]. The static strategies that their work considers are: type checking, conventional static analyses (e.g. control flow), model checking, and program verification. Salient among the surveyed works is Liebig et al.'s who propose an approach for variability-aware type checking and liveness analysis that enhances traditional static analyses and AST representations with variability knowledge stemming from annotations in C code and variability models, for example, derived from configuration files [39]. Along the same lines, Cafeo et al. propose an approach that employs a clustering algorithm to segregate members of feature interfaces that are relevant for maintenance tasks from those that are not [40]. In stark contrast with all these works our focus is on measuring source level features and their interactions. Related to this topic is the work of Dit et al. that provide a survey and taxonomy of feature location [41]. However, their work is not focused on variable software, where feature location entails considering that features can be present in different combinations to form a typically large number of different variants.

Variable Software Testing. There is an extensive and recent interest in the area of testing variable software as attested by several systematic mapping studies (e.g. [42], [43]). Salient among the identified techniques was *Combinatorial Interaction Testing (CIT)*, that when applied to variable software systems selects a set of system variants that contain all possible combinations of a given number t of selected and unselected features. The vast majority of approaches focuses only on computing the samples of variants to test based purely on variability models (e.g. feature models) without considering how the distinct feature combinations are actually implemented. In other words, they are oblivious to how features interact when they are realized [16]. Another related common thread in the surveyed approaches is that they do not empirically show that higher order interactions (i.e. $t > 3$) are more effective for fault detection to actually pay off for their typically more expensive computation. Exploiting the information obtained in our study can lead to the computing of more accurate covering arrays that do not test combinations of features that do not interact, hence significantly reducing the overall testing effort. For instance, a software engineer could employ our prediction to reduce the number of interactions for $t = 3$, by predicting them from known interactions with $t = 2$.

VIII. Conclusions

In this paper we investigated the possibility of reducing the number of feature interactions that have to be handled when managing variable software systems. With interactions of one order extracted from source code and a simple prediction, we were able to significantly reduce the number of feature interactions of the next higher order that must be considered. Moreover, we found that our results can be further enhanced by introducing a threshold of how often an interaction has to be predicted in order to be included in the results. These findings are relevant in the context of numerous different ongoing challenges when working with variable software systems (e.g. combinatorial explosion when testing).

IX. Future Work

We now sketch the most important items of our future work. First, we plan to expand our empirical study to consider other programming languages, other variability mechanisms, and consequently more examples of publicly available variable systems, to further explore the impact of these factors. Second, we plan to go beyond structural interactions to include data flow and control flow analysis. Our goal is to investigate if the prediction also works for other kinds of feature interactions, or even if a prediction of interactions over different kinds is possible. Third, we consider it a worth while endeavor to apply our prediction to real life applications, like Combinatorial Interaction Testing (CIT). We are interested if we could possibly reduce the test effort, by excluding unnecessary interactions.

Acknowledgments

Stefan Fischer is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute for Software Systems Engineering. This work was partially supported by Pro^{2}Future (FFG, Contract No. 854184) and the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN-2017-05421.

References

[1] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[2] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806819

[3] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Addison-Wesley, 2002, vol. 3.

[4] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 355–371, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2004.23

[5] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 391–400. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.61

[6] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, 2013, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2491627.2491644

[7] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, Dec 2013.

[8] P. Zave, "Faq sheet on feature interaction," http://www.research.att.com/ pamela/faq.html.

[9] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski, "Cool features and tough decisions: a comparison of variability modeling approaches," in *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, U. W. Eisenecker, S. Apel, and S. Gnesi, Eds. ACM, 2012, pp. 173–182. [Online]. Available: http://doi.acm.org/10.1145/2110147.2110167

[10] S. Apel, S. S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin, "Exploring feature interactions in the wild: the new feature-interaction challenge," in *5th International Workshop on Feature-Oriented Software Development, FOSD '13, Indianapolis, IN, USA, October 26, 2013*, A. Classen and N. Siegmund, Eds. ACM, 2013, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2528265.2528267

[11] S. Apel, J. M. Atlee, L. Baresi, and P. Zave, "Feature interactions: The next generation," Dagstuhl Seminar, Tech. Rep. 14281, July 2014. [Online]. Available: http://www.dagstuhl.de/14281

[12] D. S. Batory, P. Höfner, and J. Kim, "Feature interactions, products, and composition," in *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, E. Denney and U. P. Schultz, Eds. ACM, 2011, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2047862.2047867

[13] S. Apel, A. von Rhein, T. Thüm, and C. Kästner, "Feature-interaction detection based on feature-based specifications," *Computer Networks*, vol. 57, no. 12, pp. 2399–2409, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2013.02.025

[14] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE, 2012, pp. 167–177. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227196

[15] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE-28*. ACM, 2006, pp. 112–121.

[16] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed, "A first systematic mapping study on combinatorial interaction testing for software product lines," in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2015.7107435

[17] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba, *Computational Intelligence and Quantitative Software Engineering*. Springer, 2015, ch. Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges, accepted for publication.

[18] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: a survey and directions for future work," in *SPLC*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2648511.2648513

[19] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008. [Online]. Available: https://doi.org/10.1109/MITP.2008.54

[20] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491436

[21] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 650–670, 2014. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2014.2327020

[22] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed, "A systematic mapping study of search-based software engineering for software product lines," *Journal of Information and Software Technology*, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2015.01.008

[23] T. Berger, D. Lettner, J. Rubin, P. Grnbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *Proceedings 19th Int'l Software Product Line Conference*, ser. SPLC'15, 2015.

[24] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001

[25] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Recovering traceability between features and code in product variants," in *SPLC*, T. Kishi, S. Jarzabek, and S. Gnesi, Eds. ACM, 2013, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/2491627.2491630

[26] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "A source level empirical study of features and their interactions in variable software," in *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*. IEEE Computer Society, 2016, pp. 197–206. [Online]. Available: https://doi.org/10.1109/SCAM.2016.16

[27] "Java mpeg player," http://peace.snu.ac.kr/dhkim/java/MPEG/, accessed: 2015-08-20.

[28] R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal, and A. Egyed, "From requirements to features: An exploratory study of feature-oriented refactoring," in *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, Eds. IEEE, 2011, pp. 181–190. [Online]. Available: http://dx.doi.org/10.1109/SPLC.2011.52

[29] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *CSMR*, 2011, pp. 191–200.

[30] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: static analyses and empirical results," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 140–151. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568283

[31] "CLOC: Count lines of code," http://cloc.sourceforge.net/, accessed: 2015-08-20.

[32] "Sat4j: the boolean satisfaction and optimization library in java," http://www.sat4j.org/, accessed: 2015-08-20.

[33] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.

[34] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. [Online]. Available: https://doi.org/10.3102/10769986025002101

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29044-2

[36] T.-W. Loong, "Understanding sensitivity and specificity with the right side of the brain," *BMJ*, vol. 327, no. 7417, pp. 716–719, 2003. [Online]. Available: http://www.bmj.com/content/327/7417/716

[37] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, May 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1486

[38] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, 2014. [Online]. Available: http://doi.acm.org/10.1145/2580950

[39] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 81–91. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491437

[40] B. B. P. Cafeo, C. Hunsen, A. Garcia, S. Apel, and J. Lee, "Segregating feature interfaces to support software product line maintenance," in *MODULARITY*, vol. TBD, 2016, pp. 1–34. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9360-1

[41] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013. [Online]. Available: http://dx.doi.org/10.1002/smr.567

[42] E. Engström and P. Runeson, "Software product line testing - a systematic mapping study," *Inform. & Software Tech.*, vol. 53, no. 1, pp. 2–13, 2011.

[43] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information & Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[44] B. Meyer, L. Baresi, and M. Mezini, Eds., *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2491411