

The role of similarity in detecting feature interaction in software product lines

by

Seyedehzahra Khoshmanesh

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Robyn R. Lutz, Major Professor
Myra B. Cohen
James Lathrop

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

Copyright © Seyedehzahra Khoshmanesh, 2019. All rights reserved.

DEDICATION

To Dad, who showed me the beauty of nature and to my husband and my son without whose support I would not have been able to complete this work. I would also like to thank my advisor who guided me in this process and the committee who kept me on track.

TABLE OF CONTENTS

	Page
LIST OF TABLES	iv
LIST OF FIGURES	v
ACKNOWLEDGMENTS	vi
ABSTRACT	vii
CHAPTER 1. INTRODUCTION	1
CHAPTER 2. REVIEW OF LITERATURE	5
CHAPTER 3. METHODS AND PROCEDURES	7
3.0.1 Similarity Framework	7
3.0.2 Similarity Measures	8
3.0.3 Algorithms	11
3.0.4 SPL Case Studies	15
3.0.5 Illustrative Example	16
CHAPTER 4. RESULTS	21
4.0.1 RQ1	22
4.0.2 RQ2	24
4.0.3 Threats to Validity	26
CHAPTER 5. CONCLUSION	32
BIBLIOGRAPHY	33

LIST OF TABLES

	Page
Table 3.1	Similarity metrics used in this thesis for small SPLs 9
Table 3.2	Overview of Software Product Lines investigated in this thesis 16
Table 3.3	Known feature interactions, with their identifiers from the original sources . 17
Table 3.4	Summary of feature interaction detection of our model with different similarity metrics 18
Table 3.5	Summary of feature interaction detection using Jaccard and Hamming distance 19
Table 4.1	Details of unwanted feature interaction detection across class-based SPL case studies: Email, Elevator, MinePump 22
Table 4.2	Accuracy and Coverage of our method across the three class-based SPL case studies: Email, Elevator and Mine Pump 25
Table 4.3	Accuracy and Coverage in two different threshold settings of the similarity algorithm 25

LIST OF FIGURES

	Page
Figure 1.1 Feature Interaction Detection Using Similarity in Software ProductLines . .	3
Figure 3.1 Feature Interaction Detection Using Similarity in Software Product Lines .	8
Figure 3.2 A small software product line case study: Electronic Email System	16
Figure 3.3 An example of the process performed by our similarity framework to predict a new feature interaction	18
Figure 4.1 Jaccard Distances between features in the Email SPL case study	23
Figure 4.2 Hamming Distances between features in the Email SPL case study	24
Figure 4.3 Similarity calculations identify Encrypt and Autoresponder are highest sim- ilar features to Verify and Decrypt respectively	28
Figure 4.4 Similarity calculations identify Encrypt and Autoresponder are highest sim- ilar features to Verify and Decrypt respectively	29
Figure 4.5 Jaccard Distances between features in the Elevator SPL case study	29
Figure 4.6 Hamming Distances between features in the Elevator SPL case study	30
Figure 4.7 Jaccard and Hamming Distances between features in the MinePump SPL case study	30
Figure 4.8 Hamming Distances between features in the MinePump SPL case study . .	31

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation Grant CCF 1513717. I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Robyn R. Lutz for her guidance, patience and support throughout this research and the writing of this thesis. Her insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Myra B. Cohen and Dr. James Lathrop.

ABSTRACT

As a software product line evolves, it typically introduces new features and includes new products over time. A feature is a unit of functionality. Unwanted feature interactions, wherein one feature hinders another features operation, are a significant problem, especially as large software product lines evolve. This can lead to failures, performance degradation, and hazardous states in a new product. Software product line developers currently identify new, unwanted feature interactions primarily in the testing of each new product. This incurs significant costs, comes late in development, and does not exploit the knowledge of prior feature interactions within a product line. The contribution of this thesis is to leverage knowledge of prior feature interactions in a software product line, together with similarity measures between the features in known feature interactions and the new features, in order to detect similar feature interactions in a new product much earlier in the development process. Results obtained from application of our approach to three small software product lines from the literature showed an accuracy of 69% to 73% and coverage of 71% to 82% in detecting feature interactions. This indicates that the use of similarity measures between features in a software product line can help detect potential feature interactions in the design phase of a newly added product.

CHAPTER 1. INTRODUCTION

A software product line (SPL) is a family of software products that share a set of basic features as a core and differ in other alternative or optional features [1]. In a SPL we can take advantage of feature modularity and combine features in various configurations to create a growing set of new products [1].

Software product lines are widely used in industry to reduce the cost and time-to-market of new products. A *feature* is defined in a software product line as a unit of functionality that provides service to users [2, 3, 4] (i.e., different from a feature in machine learning or statistics). In a software product line, features are combined in various configurations to form a growing set of new products [1, 5]. Examples include aerospace systems, train control, embedded medical devices, automotive systems, and mobile phones [6].¹

To better understand the features in a domain, developers use feature models [9]. The Feature-Oriented Domain Analysis (FODA) defines the first graphical representation of a tree to express the commonalities and variations in a software product line. The basic term in the FODA is the *feature*, defined as “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems,” [10]. In a feature model, features and their relationships are drawn in a tree diagram. The root of the tree is the product line. Every node in the tree is optional or mandatory [11].

Feature interaction in a software product line refers to a situation in which individual features separately behave as expected, but when merged in a new product, one or more no longer operates as desired and may even be dangerous [12]. Batory et al. describe a classical example of unintended feature interaction, attributed to Kang. A building with a fire-control feature has sensors that, when they detect fire, activate water sprinklers. A building with a flood-control feature has water sensors that, when they detect standing water, turn off the water main. Either feature operates correctly

¹An earlier version of our work described here in [7, 8].

alone; however, if the flood-control feature is added to a building having the fire-control feature, the features interfere with each other and create a hazardous situation [13].

The feature interaction problem is a challenging one that hinders the development of dependable product lines [3, 14]. As the number of features grows, the problem typically grows.

Potential unwanted feature interactions can increase exponentially with the number of features, making the task even more difficult. In safety-critical systems, unplanned feature interactions can be hazardous. Detecting unwanted feature interactions is difficult because unwanted feature interactions are emergent, implicit, and often not found until operation.

Existing approaches to the feature-interaction problem are mostly based on testing [15]. However, this approach can only detect unwanted feature interactions late in the development process after the product has already implemented. Earlier approaches using model checking have been proposed; however, are difficult to apply in practice and are not scalable to actual SPLs [16, 17, 18].

The work reported in this thesis leverages knowledge of prior unwanted feature interactions in a product line, together with similarity measures, to improve detection of unwanted feature interactions in a new product in a software product line. We calculate similarity among features based on the degree of diversity among the features' class attributes and methods. While previous approaches can detect unwanted feature interactions in the testing phase, we want to detect them earlier, in the design phase.

Our approach uses the fact that product line repositories typically include documentation of known unwanted feature interactions, derived from previous experience and bug reports. We use similarity measures to calculate the similarity between those features known to interact in the software product line and the new features. We then employ this information to build a model that detects similar unwanted feature interactions in the new product much earlier in the development process. Figure 1.1 shows our approach, which is described in chapter 3.

We thus target two goals in our thesis. First, we want to understand whether information about software product line features' structural elements can suffice to detect potential new unwanted

feature interactions. Second, we want to investigate whether similarity measures can help achieve this detection. To address these issues, the thesis aims to address the following questions:

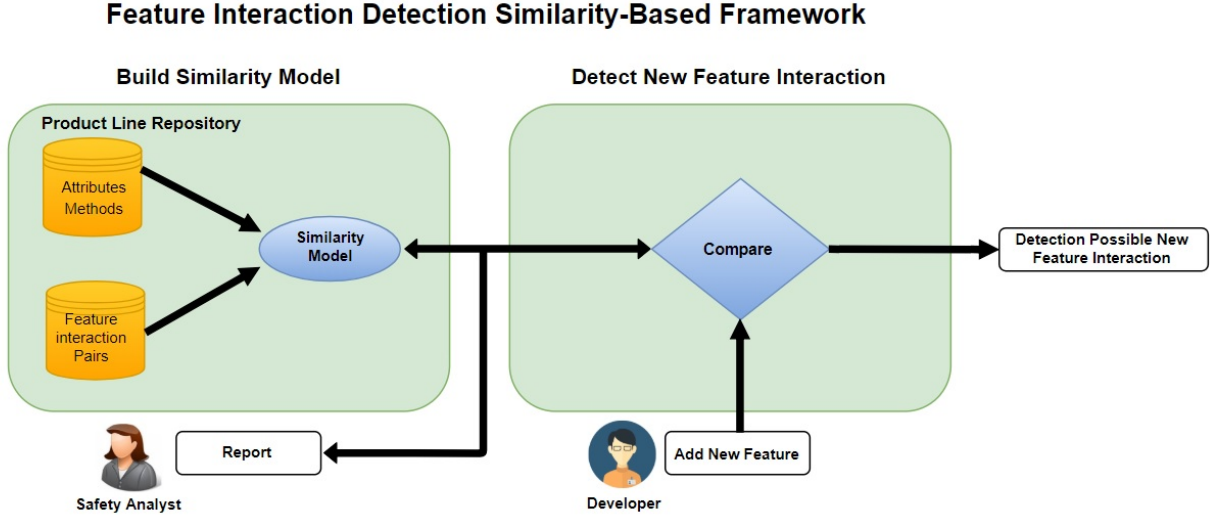


Figure 1.1: Feature Interaction Detection Using Similarity in Software ProductLines

- **RQ1:** *How effectively can we measure the similarity between a new feature and existing features using structural similarity measures?*
- **RQ2:** *To what extent does a high similarity measure between a new and an existing feature, in the context of a known unwanted feature interaction, detect possible unwanted feature interactions in a new product?*

We investigate these research questions by applying our approach to three case studies: Electronic Mail system introduced by Hall in [19] and extended as a benchmark in software product line literature [16], Elevator, and Mine Pump [16].

Results obtained from the application of our approach to three software product line case studies reported here showed an accuracy of 69% to 73% and coverage of 71% to 82% in detecting unwanted feature interactions. This indicates that the use of similarity measures between features

in a software product line can help detect potential unwanted feature interactions in the design phase of a newly added product.

The contribution of the thesis is a framework to detect unwanted feature interactions in the early-phase development of a new product in a software product line, based on the structural similarity model at the product line's feature level. While similarity measures have been considered widely in software testing, to our knowledge they have not been studied for detection of unwanted feature interactions at the feature level for a new product in a software product line.

CHAPTER 2. REVIEW OF LITERATURE

In the area of using similarity measures in a software product line, Henard, et al. [15] used similarity measures to prioritize test cases in order to reduce the number of product configurations in software product-line testing. This thesis is different in that we used similarity measures to detect feature interaction at the design stage. To our knowledge similarity measures have not been studied in order to detect feature interactions in a new product of a software product line.¹

Atlee, Fahrenberg and Legay [20] used simulation on formal models (featured transition systems) to measure the degree to which a product’s behavior differs when a new feature is added. Unlike us, they did not distinguish between intended and unintended feature interactions.

Soares, et al. [3] performed a recent systematic study of works on feature interaction in software product line engineering. They found that 43% of the papers aimed to understand feature interaction at early stages of the software life cycle. Among this 43%, the majority used formal methods, specifically model checking.

Apel, et al. [12, 21] proposed feature-aware verification to automate the detection of feature interactions using variability encoding. Our approach to dealing with the feature interaction problem differs from their studies in not requiring formal methods, motivated by the current low uptake of formal methods by industries developing software product lines.

Al-Hajjaji, et al. [22] proposed a similarity-based prioritization that increases coverage of SPL test cases to detect errors in reasonable time. They compared the result of their algorithm with three sampling algorithms and concluded that the similarity-based prioritization algorithm can compete with them and produce the test cases faster.

Sánchez, Segura, and Ruiz-Cortés [23] investigated five different prioritization criteria including dissimilarity to generate test cases for software product-line testing. They obtained 87% accuracy with prioritization based on dissimilarity. While we use a similarity model to detect feature inter-

¹An earlier version of our work described here in [7], [8].

action, their work differs from ours in that we apply similarity to individual features rather than to the entire product in a SPL and detect feature interaction at the design stage rather than the testing phase.

Sahak, Jawawi, and Halim [24] also evaluated different similarity measures for test case prioritization in software product lines. The authors found that a text matching measure performed best for test case prioritization in the SPL. However, this thesis is very different since we use similarity measures on product-line features rather than on SPL test cases.

We instead aim in our method to leverage feature similarity to detect unwanted feature interactions in an evolving software product line, since our method is more understandable to the developers and users who work with and maintain the product-line systems.

CHAPTER 3. METHODS AND PROCEDURES

Our work aims to discover unwanted feature interactions in a new SPL product at an earlier stage of the new product’s development. To this end, we have proposed a method that leverages knowledge of prior unwanted feature interactions in the SPL together with measures of similarity between existing and new features. This enables design-time discovery of unwanted feature interactions in a new product.¹

Our approach uses the fact that product line repositories typically include documentation of known unwanted feature interactions, derived from previous experience and bug reports. We use similarity measures to calculate the similarity between those features known to interact in the software product line and the new features. We then employ this information to build a model that detects similar unwanted feature interactions in the new product much earlier in the development process. As detecting feature interaction at the code level is partial, costly, and occurs late in development, we concentrate on the early stage artifacts of an SPL consisting of the feature models and the class elements related to each product line feature. Hence, we introduce an efficient framework using similarity measures to detect new unwanted feature interactions in evolving software product lines.

3.0.1 Similarity Framework

Our approach uses two main information sources to detect unwanted feature interactions: (1) the SPL repository of known unwanted feature interactions and a set of early-stage artifacts, i.e., the feature model, class attributes and methods; and (2) a set of similarity measures to understand how close new features are to existing features. The proposed framework shown in Figure 1.1 and duplicated here in Figure 3.1.

¹An earlier version of our work described here in [7], [8].

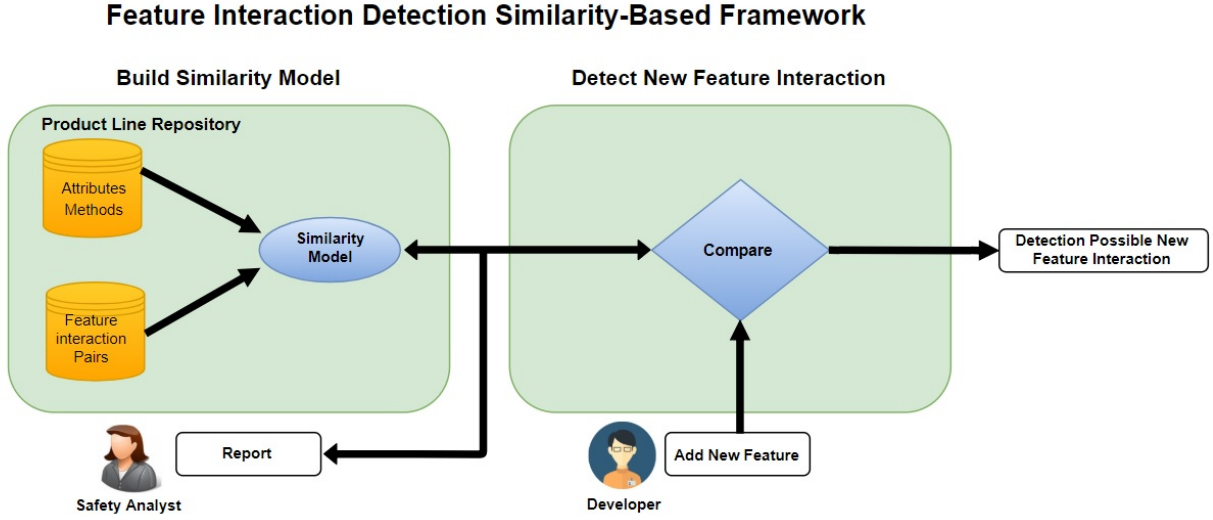


Figure 3.1: Feature Interaction Detection Using Similarity in Software Product Lines

As displayed there, known unwanted feature interactions are derived from bug reports for earlier products and from early-stage elements of product line features such as feature models, as well as class variables and class methods, if available in the repository.

A proposed usage scenario for our similarity framework is that a developer wants to understand whether a new product having a new feature F_{new} will interact in an undesired way with any existing features. We use the SPL artifacts with its capture of known interactions to help answer this question by applying feature similarity measures to F_{new} and features in known and unwanted interactions. The approach reports to the developer the extent to which the new feature F_{new} may potentially participate in any known feature interaction. This technique has the potential to reduce feature-interactions inadvertently introduced in a new product, thereby reducing risk as well as saving time and effort in debugging.

3.0.2 Similarity Measures

The similarity is the pivotal piece of our framework. We use similarity as a heuristic tool to compare two features in the SPL. Table 3.1 shows the similarity measures applied in the thesis for

computing the similarity score of two features in a small SPL. We selected two set-based similarity measures, Jaccard [25] and Hamming [26], to obtain a similarity score between two features when we have access to the SPL class diagram including class variables and methods. We represent a feature as a set of class methods and class variables used in the design of a feature. Therefore, the feature representation is a class-based artifact since only methods and variables of a coded feature are considered to capture the properties of a feature in an SPL.

Table 3.1: Similarity metrics used in this thesis for small SPLs

Name of Metric	Category	Case Studies	Ref.
Jaccard	Class	Email,Elevator,MinePump	[25]
Hamming	Class	Email,Elevator,MinePump	[26]

We represent a feature as a set of class methods and class variables used in class implementation of a feature. Therefore, the feature representation is a class-based artifact since only methods and variable of a coded feature are considered to capture the properties of a feature in an SPL. In similar literature in SPL testing, researchers have successfully employed both Jaccard and Hamming distances to prioritize product line test cases [15, 27, 23, 22]. These two similarity metrics are used in the software testing literature to compare two products, while we use them here to compare two features in an SPL. Jaccard and Hamming are defined as follows:

Jaccard distance is a set-based metric for measuring the similarity value of two sets. In this thesis, each feature in a small SPL has a set of class variables and methods. We compare the sets of class variables and methods between pairs of features. Let F be a feature in an SPL. Jaccard measure is defined as follows in equation (3.1):

$$J : F \times F \longrightarrow [0, 1] \tag{3.1}$$

$$J(Fi, Fj) = \frac{|Fi \cap Fj|}{|Fi \cup Fj|}, \quad \text{Where } Fi, Fj \in F$$

The value range of the Jaccard distance is 0 to 1. 0 indicates that two features in an SPL are completely different, while 1 means that these two features are equal. $|Fi \cap Fj|$ points to the number of common class variables and methods of two product-line features. $|Fi \cup Fj|$ refers to the number of members of two Product Line (PL) features. As an example of how Jaccard distance is computed between two features:

Jaccard Example

$$F1 = \{a, b, c, d\}, F2 = \{a, c, f\}$$

$$|F1 \cap F2| = |\{a, c\}| = 2$$

$$J(F1, F2) = \frac{|F1 \cap F2|}{|F1 \cup F2|} = \frac{2}{5} = 0.4$$

In this example, $F1$ is represented as set of four values a, b, c, d . These values can be artifacts which form the feature, here class methods and variables of a feature. $F2$ also is defined with a set of three values a, c, f . The conjunction or intersection of these two features is a set having two values a, c . The disjunction or union of feature $F1$ and $F2$ is a set having five members, a, b, c, d, f . Thus, the Jaccard distance score of these two features is the size of the first set divided by the size of the second set that is 0.4.

Hamming distance is a set-based metric which measures the similarity between two sets, here two features in an SPL. Hamming distance is given by equation (3.2). Following Al-Hajjaji, et al. [22], we also evaluate Hamming distance as a measure of similarity in our applications.

$$H : F \times F \longrightarrow [0, 1] \tag{3.2}$$

$$H(Fi, Fj) = \frac{|Fi \cap Fj| + |(S \setminus Fi) \cap (S \setminus Fj)|}{|S|},$$

$$\text{Where } Fi, Fj \in F, S = \bigcup_{i \in I} Fi, I = \{1, \dots, n\}$$

The explanation of similarity incorporated in Hamming distance is the same as that in Jaccard distance. The similar features have the Hamming score close to 1. In the Hamming formula, deselected members of two sets, here class variables and methods, are reflected as well. Deselected members of the two sets are taken into account because some interactions take place due to the deselection of the set's members [22, 28]. As an example of how to compute Hamming distance between two features:

Hamming Example

$$\begin{aligned}
 F1 &= \{a, b, c, d\}, F2 = \{a, c, f\}, F3 = \{b, e\} \\
 S &= \bigcup_{i \in I} Fi = F1 \cup F2 \cup F3 = \{a, b, c, d, e, f\} \\
 |F1 \cap F2| &= |\{a, c\}| = 2 \\
 |(S \setminus F1) \cap (S \setminus F2)| &= |\{e\}| = 1 \\
 H(F1, F2) &= \frac{|F1 \cap F2| + |(S \setminus F1) \cap (S \setminus F2)|}{|S|} = \frac{2+1}{6} = 0.5
 \end{aligned}$$

3.0.3 Algorithms

Algorithm 1 and 2 shows how we calculate the Jaccard and Hamming between two features in an SPL. Algorithm 3 describes how we compute the Pairwise calculation of similarity scores between features in SPL case studies in our similarity-based detection framework. Finally, Algorithm 4 explains how we recommend new potential feature interaction when a new feature is added to the SPL. We implemented these four algorithms in Python and used them to obtain the results described in Chapter 4.

Algorithm 1 Calculate Jaccard Similarity

Input: $Fi, Fj \in F$ (Two Feature Sets)

Output: $JSim$ (Jaccard Similarity)

```

1: function JACCARD( $Fi, Fj$ )
2:    $Union \leftarrow |Fi \cup Fj|$ 
3:    $Intersection \leftarrow |Fi \cap Fj|$ 
4:    $JSim \leftarrow \frac{Intersection}{Union}$ 
5:   return  $JSim$ 
6: end function

```

Algorithm 2 Calculate Hamming Similarity

Input: $Fi, Fj \in F, S = \bigcup_{i \in I} Fi, I = \{1, \dots, n\}$
Output: $HSim$ (Hamming Similarity)

```

1: function HAMMING( $Fi, Fj, S$ )
2:    $Union_{All} \leftarrow S$ 
3:    $Intersec_{Fi, Fj} \leftarrow |Fi \cap Fj|$ 
4:    $Diff_{Fi} \leftarrow |(S - Fi)|$ 
5:    $Diff_{Fj} \leftarrow |(S - Fj)|$ 
6:    $Intersect_{Diff} \leftarrow Diff_{Fi} \cap Diff_{Fj}$ 
7:    $HSim \leftarrow \frac{Intersec_{Fi, Fj} + Intersect_{Diff}}{Union_{All}}$ 
8:   return  $HSim$ 
9: end function

```

Algorithm 3 Calculate Pairwise Similarity of all features in SPL

Input: $F_1 \dots F_N$

Output: $JSim[\][\], HSim[\][\]$ (Pairwise Similarity Matrix)

```

1: function SIMMATRIX( $F_1 \dots F_N$ )
2:   for  $i \leftarrow 1$  to  $N$  do
3:     for  $j \leftarrow i + 1$  to  $N$  do
4:        $JSim[i][j] \leftarrow JACCARD(F_i, F_j)$ 
5:        $HSim[i][j] \leftarrow HAMMING(F_i, F_j)$ 
6:     end for
7:   end for
8:   return  $JSim[\ ][\ ], HSim[\ ][\ ]$ 
9: end function

```

Algorithm 4 Algorithms for Detecting New Unwanted Feature Interaction

Input: $F[\]$, $FI[\]$, F_{new} (FI: list of known pairwise unwanted feature interactions in the SPL)

Output: $NewFI[\]$ (list of new suggested unwanted feature interactions)

```

1: function DETECT( $F[N]$ ,  $FI[K]$ ,  $F_{new}$ )
2:    $FSimilarJ \leftarrow F1$ 
3:    $FSimilarH \leftarrow F1$ 
4:    $JMax \leftarrow JACCARD(F1, F_{new})$ 
5:    $HMax \leftarrow HAMMING(F1, F_{new})$ 
6:   for  $i \leftarrow 2$  to  $N$  do
7:      $JSim[i] \leftarrow JACCARD(Fi, F_{new})$ 
8:      $HSim[i] \leftarrow HAMMING(Fi, F_{new})$ 
9:     if ( $JSim[i] > JMax$ ) then
10:       $FSimilarJ \leftarrow Fi$ 
11:     end if
12:     if ( $HSim[i] > HMax$ ) then
13:       $FSimilarH \leftarrow Fi$ 
14:     end if
15:   end for
16:   for  $i \leftarrow 1$  to  $k$  do
17:      $Fi + FsimilarJ \leftarrow FindFI(FsimilarJ)$ 
18:      $NewFI[] \leftarrow Fi + FsimilarJ$ 
19:      $Fi + FsimilarH \leftarrow FindFI(FsimilarH)$ 
20:      $NewFI[] \leftarrow Fi + FsimilarH$ 
21:   end for
22:   return  $NewFI[\ ]$ 
23: end function

```

Time Complexity of Algorithms: Time complexity of Algorithms 1 and 2 are constants. The time complexity of Algorithm 3 which calculates the pairwise similarity score between features in the small SPL is quadratic in the number of features in the SPL. Assume we have n features in the SPL. Algorithm 3 runs at most $n(n-1)/2$ times to calculate the similarity score of all pairwise features. Since we did not weigh order between features, a combination of $f1 + f2$ is the same as $f2 + f1$. Therefore, the total number of execution times is divided by 2. Thus, the final time complexity is $O(N^2)$.

The time complexity of Algorithm 4, to detect potential feature interaction, depends on the number of existing features participate in known unwanted feature interactions. This algorithm computes the similarity between new features and these known features which are at most n . Therefore, the worst time complexity of Algorithm 4 is the same as Algorithm 3, $n(n-1)/2$, and is quadratic.

3.0.4 SPL Case Studies

In this section, we introduce the case studies investigated in our proposed similarity framework for detecting unwanted feature interaction at the design phase for our small to medium size SPL. SPLs which are less than 9 features are defined as small [29]. We selected three software product lines to investigate our research questions: Email, Elevator, Mine Pump. These three SPL are considered to be benchmarks in software product line literature [16].

The EMail system was originally presented by Hall [19] models an e-mail communication system having several optional features that can be enabled or disabled such as encryption, forwarding, and verify in the email.

Plath and Ryan [30] developed the Elevator system. It models an elevator with several optional features such as that when the elevator is empty, it stops and when the elevator is full, it does not respond to the requests outside the elevator.

The MinePump system simulates a water pump in a mining operation and was described by Kramer, Magee, Sloman, and Lister [31]. The system has several features that can vary its be-

havior. The pump keeps the bottom of the mine shaft dry, but must be deactivated when the mine contains combustible methane gas. We used the versions of these three software product line developed by Apel et al. [32, 16, 12].

Table 3.2: Overview of Software Product Lines investigated in this thesis

SPL Name	$ Features $	$ FeatureInteractions $	<i>Language</i>	<i>#LoC</i>	<i>Refrence</i>
<i>Elevator</i>	6	6	Java	1046	[16]
<i>MinePump</i>	7	4	Java	580	[16]
<i>Email</i>	9	11	Java	1233	[16]

3.0.5 Illustrative Example

In this section, we explain how our similarity framework detects unwanted feature interactions in one of our case studies, the Electronic mail system extracted from [19, 16, 21]. The electronic email system implements an Email system which shares a base Email Client and has seven optional features. Figure 3.2 shows the feature diagram of the Email system. Moreover, there are eleven known unwanted feature interactions in the Email case study obtained from [33, 19, 21]. These are shown in Table 3.3.

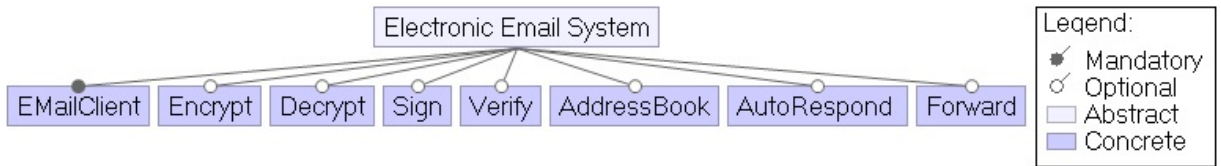


Figure 3.2: A small software product line case study: Electronic Email System

We next explain the interaction between Encrypt and Forward and describe how our similarity framework detects new feature interaction, and refer the reader to [33] and [19] for the details of the other interactions shown in Table 3.3. The unwanted feature interaction happens when EmailClient

Table 3.3: Known feature interactions, with their identifiers from the original sources

Feature Id	Features Involved
0	Decrypt , Forward
1	Addressbook ,Encrypt
3	Sign ,Verify
4	Sign , Forward
6	Encrypt , Decrypt
7	Encrypt , Verify
8	Encrypt , Autoresponder
9	Encrypt , Forward
11	Decrypt , Autoresponder
13	Autoresponder , Forward
27	Verify , Forward

sends an encrypted email to a second email client, and asks the second email client to forward the email to a third party. The second host sends the email in plain text if the public key of the third party is not available. This scenario can be hazardous since it violates the security property of the system that states that encrypted email must remain encrypted over the network.

Figure 3.3 displays the process performed by our framework to detect new unwanted feature interactions. Consider feature interaction with Id=8 in Table 3.3 between Autoresponder and Encrypt. (As a reminder, the EmailClient feature must be in all products created in the E-mail SPL.) We consider other features such as Addressbook, Decrypt, Forward, Sign, and Verify as possible new features that we would like to add. Next, we measure the Jaccard and Hamming similarity between new features and features in the known interaction, here between Autoresponder and Encrypt. As depicted in Figures 4.1, Decrypt has the highest similarity to the Autoresponder in both Jaccard distance and Hamming distance. Therefore, our similarity framework detects the pair Decrypt-Encrypt as a potential feature interaction. This is a true detection [33, 19] meaning that there is a real unwanted feature interaction between these two features.

<i>Known Feature Interaction</i>	Encrypt, Autoresponder (8)		
<i>Possible new features</i>	Addressbook, Decrypt, Forward, Sign, Verify		
	Encrypt		Autoresponder
<i>Similar features by Jaccard</i>	Verify		Decrypt
<i>Jaccard Detection</i>	Verify-Autoresponder		Encrypt-Decrypt
<i>True Detection</i>	✗		✓
<i>Similar feature by Hamming</i>	Decrypt Verify		Decrypt
<i>Hamming Detection</i>	(Decrypt – Autoresponder)	(Verify – Autoresponder)	Encrypt-Decrypt
<i>True Detection</i>	✓	✗	✓

Figure 3.3: An example of the process performed by our similarity framework to predict a new feature interaction

We use this process for all known unwanted feature interactions in the SPL repository. Safety analysts and developers then can use the output from our similarity framework to recognize possible unwanted feature interactions in the new product without waiting until the test state to identify them. In comparison with detecting unwanted feature interactions by testing, our approach is less expensive and able to detect unwanted feature interactions earlier at the design phase when developing a new product in the SPL.

Table 3.4: Summary of feature interaction detection of our model with different similarity metrics

	True Detection	False Detection	Accuracy
Jaccard	16	7	70%
Hamming	19	7	73%
Combined	21	9	70%

As depicted in Table 3.4, the model using Jaccard detected 23 pairs as unwanted feature interactions of which 16 are true detection and 7 are false positives. Our similarity model using Hamming detected 26 pairs as unwanted feature interactions shown in Table III. While 19 pairs were detected correctly as unwanted feature interactions, 7 pairs were false alarms.

The total detection using the Hamming model was slightly more than using the Jaccard model since in some cases there was more than one feature having the highest similarity. Thus, we reported all features having the highest similarity as candidates for feature interaction.

We next combined Jaccard and Hamming and repeated the experiment. The performance results under the combined metric are shown in Table 3.4. The model detected 21 pairs correctly as feature interaction while 9 pairs were detected incorrectly.

Based on the results described above, we conclude that high similarity measures between features can predict new unwanted feature interactions. Although the accuracy of our proposed framework differs with different similarity metrics, the difference was small. Among 11 unwanted feature interactions shown in Table 3.5, Jaccard distance detected 10 of them while Hamming was able to detect 9 unwanted feature interactions.

Table 3.5: Summary of feature interaction detection using Jaccard and Hamming distance

FI ID	Detected by Jaccard	Detected by Hamming
0	✓	✓
1	X	X
3	✓	X
4	✓	✓
6	✓	✓
7	✓	✓
8	✓	✓
9	✓	✓
11	✓	✓
13	✓	✓
27	✓	✓
Total	10	9

Neither Hamming nor Jaccard could detect Addressbook-Encrypt as a feature interaction. This happens because Addressbook does not have the highest similarity value of any of the features in the SPL. Since it has the second or third highest value, it is not selected.

Nevertheless, if we were to allow the model to select the second or third highest similar feature, it would detect Addressbook-Encrypt and Sign-Verify as unwanted feature interactions. Investigating such alternatives is a topic for future work.

While Jaccard recognized Sign-Verify as a feature interaction, Hamming did not detect it. This is because Sign is not the feature most similar to any other feature in Hamming distance, and Hamming returned Decrypt as the most similar feature to the Encrypt feature. This causes the Sign-Verify interaction not to be detected in Hamming. To overcome this weakness we investigated the Combined metric, i.e., the disjunction of Jaccard and Hamming. The results are reported in the next Chapter.

Although the similarity model cannot detect all unwanted feature interactions, the results indicate that its use can help developers detect a significant fraction of the unwanted feature interactions in the new product in an earlier stage and at a lower cost.

CHAPTER 4. RESULTS

In this section, we present our results for each of the two research questions. We investigate two different similarity metrics in a main structure-based category, *class*. We then build and evaluate detection calculation models based on them. We applied class-based similarity metrics on Email, Elevator, and MinePump, the three SPL case studies described in Chapter 3.¹

Our goal is to detect pairwise feature interactions which means two features contribute to an interaction, i.e, the presence of one of two features causes a change in the behavior of the other feature [13]. Two-way feature interactions are the most common form of feature interactions we encounter in Software Product Line community [34, 17, 35]. Moreover, a study of analysing the variability on 40 large scale SPL showed that structural interactions exist mostly between two features [36]. Therefore, investigating the pairwise feature interactions detects the most common form of unwanted feature interactions at the early stage of a SPL [9, 13].

For RQ1, for similarity metric, we evaluate the detection model on the Electronic Mail System, the Elevator, and the MinePump SPLs. We report the performance in terms of Accuracy and Coverage of detection of unwanted feature interactions [37]. Here, *Accuracy* is defined as the number of correct unwanted feature interaction suggestions by our similarity framework divided by the total suggestions. *Coverage* is defined as the number of unique unwanted feature interactions detected by our similarity framework divided by the total unique unwanted feature interactions in the SPL. The range of value in both Accuracy and Coverage is 0 to 1. The unwanted feature interactions reported in Table 4.1 serves as the summary of all results for our SPL case studies.

¹An earlier version of our work described here in [7], [8].

Table 4.1: Details of unwanted feature interaction detection across class-based SPL case studies: Email, Elevator, MinePump

SPL Name	FI ID	Feature Involved	Detected by Jaccard	Detected by Hamming
Email	1	Decrypt - Forward	Y	Y
	2	Addressbook - Encrypt	Can be detected in 4th Level	Can be detected in 4th Level
	3	Sign - Verify	Y	Can be detected in 3rd Level
	4	Sign - Forward	Can be detected in 2nd Level	Can be detected in 2nd Level
	5	Encrypt - Encrypt	Y	Y
	6	Encrypt - Verify	Y	Y
	7	Encrypt - Autoresponder	Y	Can be detected in 2nd Level
	8	Encrypt - Forward	Y	Y
	9	Decrypt - Autoresponder	Y	Y
	10	Autoresponder - Forward	Y	Y
	11	Verify - Forward	Y	Y
Elevator	1	Overloaded - Empty	Can be detected in 3rd Level	Can be detected in 2nd Level
	2	2/3 full - Empty	Y	Y
	3	Executivefloor - Empty	Y	Y
	4	2/3full - Overloaded	Y	Y
	5	Executivefloor - Overloaded	Y	Y
	6	Executivefloor - 2/3full	Y	Y
Mine Pump	1	highWaterSensor - methaneAlarm	Y	Y
	2	highWaterSensor - methaneQuery	Can be detected in 4th Level	Y
	3	highWaterSensor - stopCommand	Can be detected in 2nd Level	Y
	4	highWaterSensor - lowWaterSensor	Y	Y

4.0.1 RQ1

How effectively can we measure the similarity between a new feature and existing features using structural similarity measures?

Our similarity-based framework computed the pairwise Jaccard and Hamming distances between all the product-line features in our small SPL case studies, the Electronic Mail System, Elevator, and MinePump.

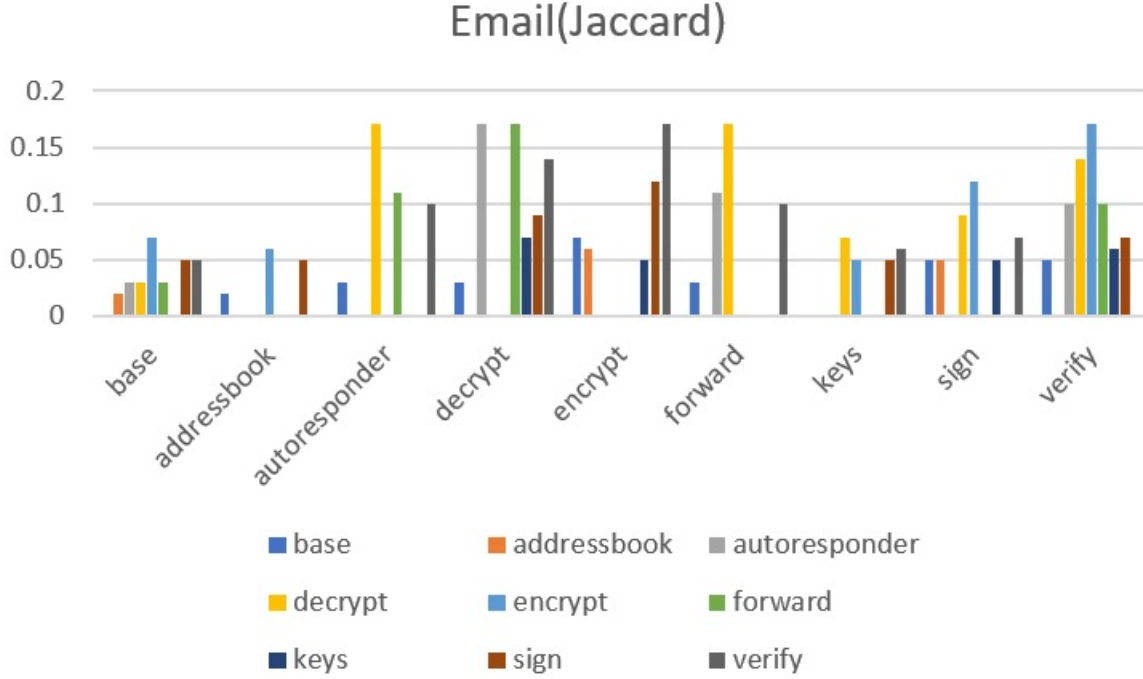


Figure 4.1: Jaccard Distances between features in the Email SPL case study

Figures 4.1 and 4.2 show the bar diagram of pairwise results for Jaccard and Hamming distances for the Email SPL case study. We set the threshold for considering two features to be similar to 0.1 for Jaccard distance and 0.8 for Hamming distance. For example, feature Decrypt is the most similar feature to the feature Autoresponder based on both Jaccard and Hamming metrics. Figures 4.3 and 4.4 shows how our similarity framework identifies two features, Decrypt and Autoresponder, similar.

Figures 4.5 and 4.6 show the bar diagram of pairwise results for Jaccard and Hamming distances for the Elevator SPL case study. Figures 4.7 and 4.8 show the bar diagram of pairwise results for Jaccard and Hamming distances for the MinePump SPL case study. Our similarity-based framework then uses these similarity scores to determine the most similar feature to the new feature.

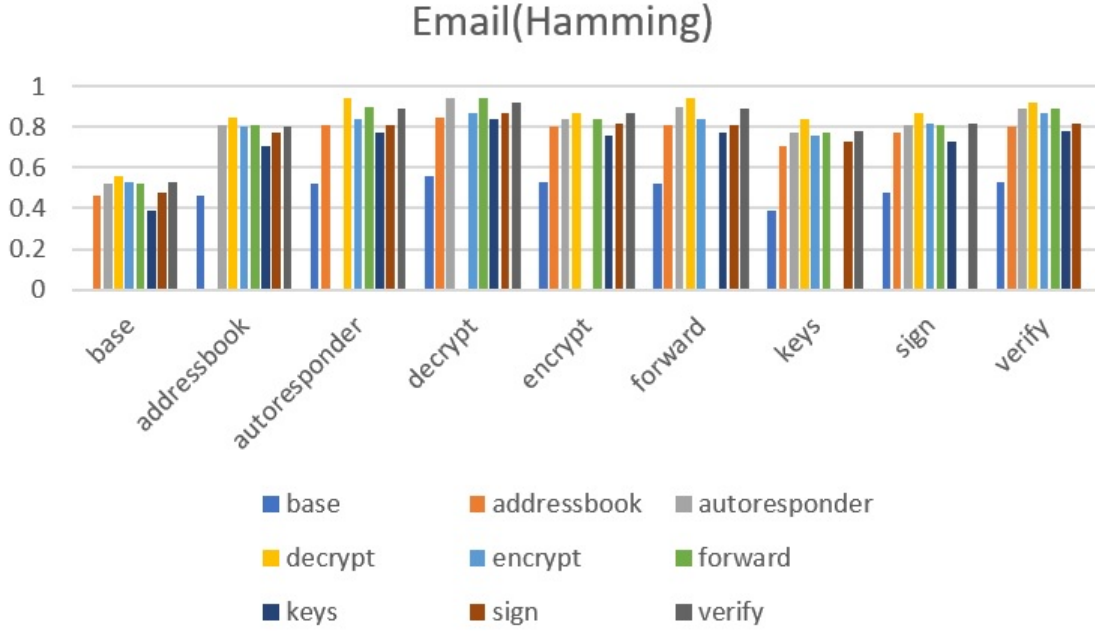


Figure 4.2: Hamming Distances between features in the Email SPL case study

Based on the result that we describe in RQ2 and Table 4.2 in terms of Accuracy and Coverage of the detection of unwanted feature interaction, we could effectively measure the similarity between a new feature and existing feature using Jaccard and Hamming distances.

4.0.2 RQ2

RQ2: To what extent does a high similarity measure between a new and an existing feature, in the context of a known unwanted feature interaction, detect possible unwanted feature interactions in a new product?

Table 4.2 shows the Accuracy and Coverage for our similarity model on Email, Elevator, and MinePump with Jaccard distance (top), Hamming distance (bottom), respectively. As shown in Table 4.2, across the three SPL case studies, the highest Accuracy obtained by Jaccard was 73.3% and the highest Coverage obtained by Hamming was 82.4%. Although the Accuracy obtained by Jaccard and Hamming are similar, the Coverage of Hamming is comparatively higher than Jaccard.

This indicates that using both Hamming and Jaccard as a combined metric, i.e., the disjunction of Jaccard and Hamming, is beneficial. This is because the combined metric takes into account features returned by Jaccard and not by Hamming, while returning the high total detection of Hamming.

Table 4.2: Accuracy and Coverage of our method across the three class-based SPL case studies: Email, Elevator and Mine Pump

	Jaccard Similarity		Hamming Similarity		Combined Similarity	
SPL Name	Accuracy	Coverage	Accuracy	Coverage	Accuracy	Coverage
<i>Email</i>	70%	82%	70%	64%	70%	82%
<i>Elevator</i>	100%	83.3%	100%	83.3%	100%	83.3%
<i>Mine Pump</i>	50%	50 %	37.5 %	100%	43.75%	100%
<i>Average</i>	73.3%	71.77%	69.17%	82.4%	71.25%	88.43%

We also investigated a relaxation version of our similarity framework in small SPLs to eliminate the constraint in our similarity framework which limits it to selecting only the highest similar feature and ignore the others. Relaxation allows the similarity framework to propose the second and third similar features as candidates for similar features, as well. Thus, we consider the second level and third level similar features as similarity candidates for new features added to the system. For example, in the Electronic Mail System as shown in Figure 4.1, the highest similar feature to Encrypt is Verify followed by Sign. We call feature Sign a second level similar feature to Encrypt.

Table 4.3: Accuracy and Coverage in two different threshold settings of the similarity algorithm

		Jaccard Similarity		Hamming Similarity		Combined Similarity	
		Accuracy	Coverage	Accuracy	Coverage	Accuracy	Coverage
E-mail	1st	70%	82%	70%	64%	70%	82%
	2nd	61%	91%	70%	82%	65.5%	91%
Elevator	1st	100%	83.3%	100%	83.3%	100%	83.3%
	2nd	100%	83.3%	100%	100%	100%	100%
Mine Pump	1st	50%	50%	37.5%	100%	43.75%	100%
	2nd	50%	75%	41%	100%	45.5%	100%

Table 4.3 shows the Accuracy and Coverage of the three SPL studies when the framework allows the second level similar feature to be selected. As shown in the table, in all cases Coverage is increased. Accuracy is increased or unchanged for Elevator and Mine Pump. This indicates considering the second level similar feature as a candidate may improve both Accuracy and Coverage, and need additional study.

Table 4.1 shows all unwanted feature interactions in the three small SPL case studies and the detection status for them. Symbol “Y” in Table 4.1 means that the unwanted feature interaction is detected by our similarity framework without any relaxation. For the unwanted feature interaction that they cannot be detected in the original algorithm, we wrote with regarding what level of relaxation, they can be detected. For example, Overload- Empty interaction can be detected in second level relaxation of our similarity algorithm with Hamming distance. In average, Hamming distance has better result compared to Jaccard in detecting the number of unique unwanted feature interactions that define as Coverage in our study. As we relaxing the algorithm, the Coverage increases while the Accuracy decreases since the framework suggests more false positive cases.

4.0.3 Threats to Validity

In this subsection we describe some threats to the validity of our model. First, we have investigated three small SPLs to evaluate our work. However, these case studies are considered to be benchmarks in the SPL literature [16, 21]. While our work can be applied to other SPLs in other domains, more work is necessary to establish the generality of this approach to larger, real-world systems.

The second main threat comes from the artifacts on which we applied our study. We do not know about their correctness. However, the case studies have been published and used by researchers, and we validated independently that the known feature interactions appeared to be complete in the context of the artifacts.

Finally, although our similarity framework uses a limited set of similarity metrics, our findings show that the accuracy of the three different similarity measures in detecting unwanted feature interaction does not differ much and that they produce similar results.

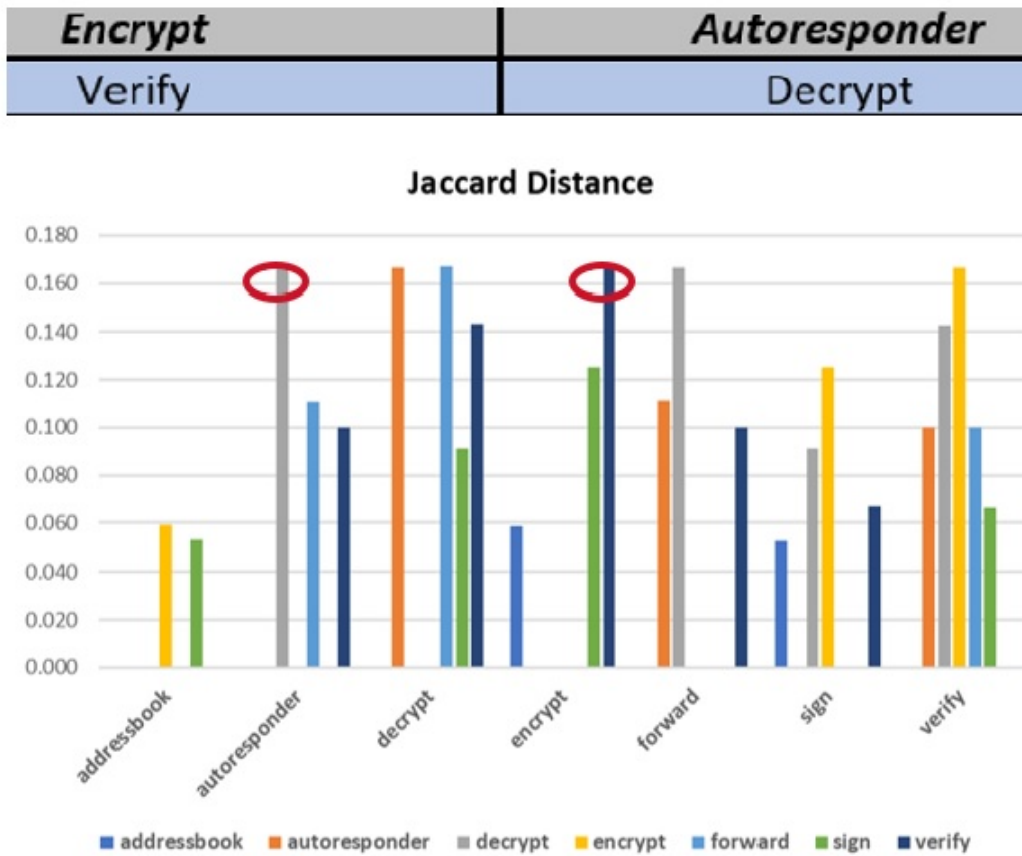


Figure 4.3: Similarity calculations identify Encrypt and Autoresponder are highest similar features to Verify and Decrypt respectively

Jaccard Distance ([0-1] ; 1:same 0:totally different)							
	addressbook	autoresponder	decrypt	encrypt	forward	sign	verify
addressbook		0.000	0.000	0.059	0.000	0.053	0.000
autoresponder	0.000		0.167	0.000	0.111	0.000	0.100
decrypt	0.000	0.167		0.000	0.167	0.091	0.143
encrypt	0.059	0.000	0.000		0.000	0.125	0.167
forward	0.000	0.111	0.167	0.000		0.000	0.100
sign	0.053	0.000	0.091	0.125	0.000		0.067
verify	0.000	0.100	0.143	0.167	0.100	0.067	

Figure 4.4: Similarity calculations identify Encrypt and Autoresponder are highest similar features to Verify and Decrypt respectively

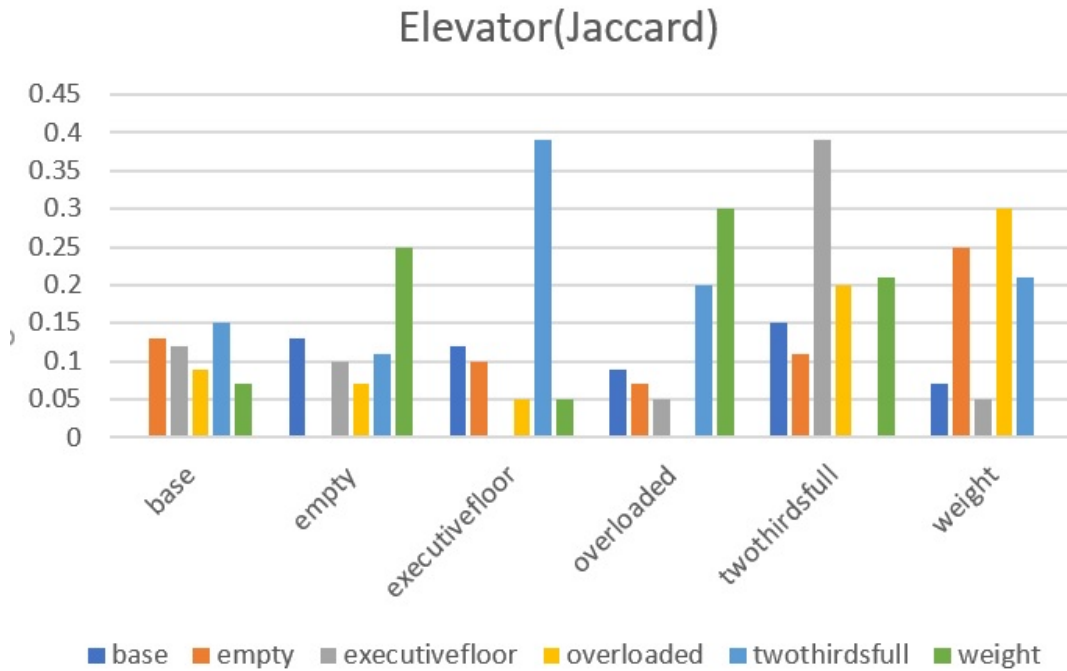


Figure 4.5: Jaccard Distances between features in the Elevator SPL case study

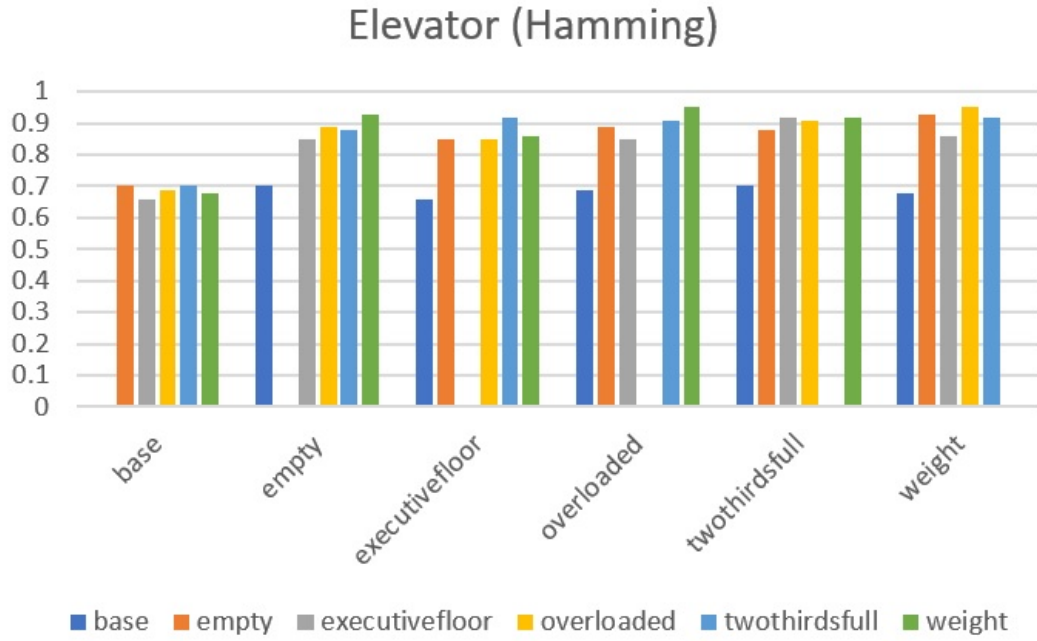


Figure 4.6: Hamming Distances between features in the Elevator SPL case study

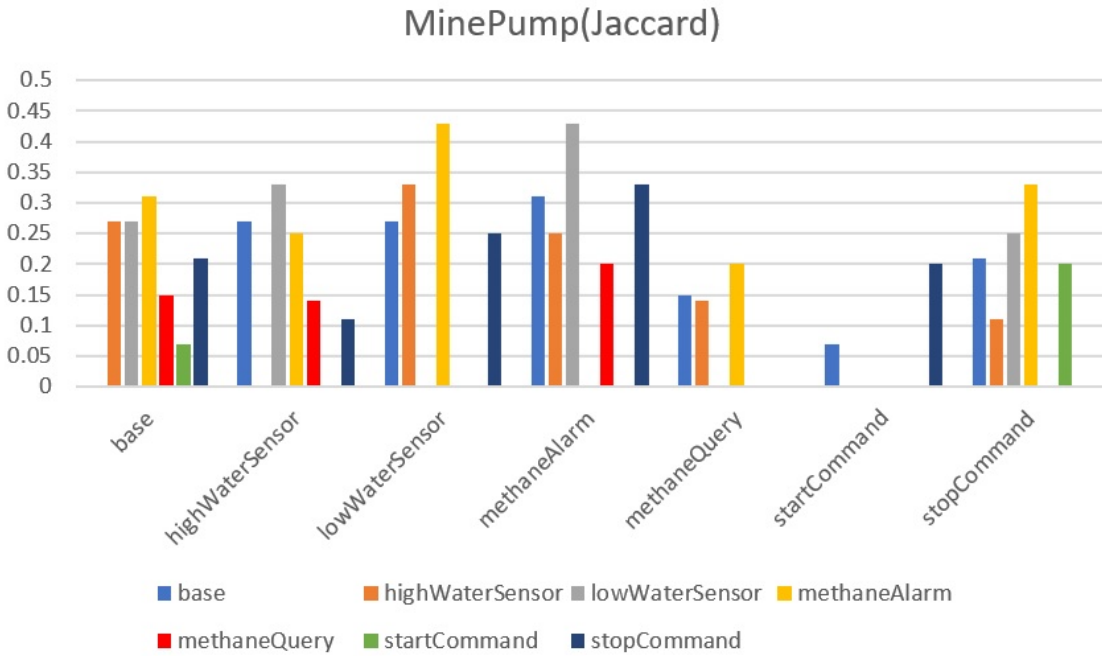


Figure 4.7: Jaccard and Hamming Distances between features in the MinePump SPL case study

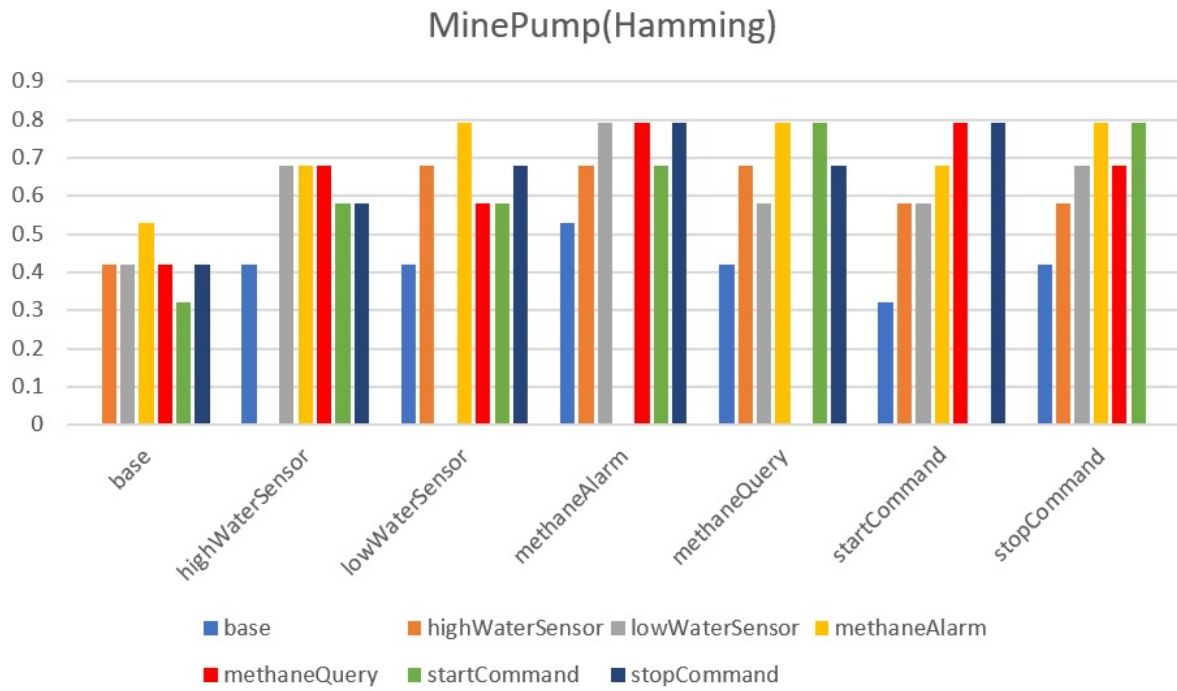


Figure 4.8: Hamming Distances between features in the MinePump SPL case study

CHAPTER 5. CONCLUSION

In this thesis we proposed a similarity-based technique to detect feature interactions at the design phase of a new product. We applied and evaluated it in three software product line using knowledge of prior feature interactions in that SPL. The heuristic behind this thesis is that similar features often behave in the same way and, if one feature is known to have a problematic interaction with a specific feature, then it is likely that a similar feature in a new product also will have a problematic interaction that specific feature, if it also is in the new product. We use structural artifacts of a product-line feature such as class attributes and methods to measure the similarity between two features.

Our results from evaluation of this approach on artifacts from three small software product lines show that using similarity measures between features in a software product line can detect possible feature interactions at the design stage.

Future work based on these results will consider more product line artifacts such as requirements documents and activity diagrams. Using additional artifacts for a software product line could lead to a more accurate model. Additionally evaluating the use of similarity measures to detect feature interactions on larger software product lines could improve understanding of similarity at scale.

BIBLIOGRAPHY

- [1] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.
- [2] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y. . Lin. The feature interaction problem in telecommunications systems. In *SETSS 89*.
- [3] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. Feature interaction in software product line engineering: A systematic mapping study. *Information and Software Technology*, 98:44–58, 2018.
- [4] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature?: a qualitative study of features in industrial software product lines. In *SPLC*, pages 16–25. ACM, 2015.
- [5] Goetz Botterweck and Andreas Pleuss. Evolution of software product lines. In *Evolving Software Systems*, pages 265–295. Springer, 2014.
- [6] Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehler, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai, editors. *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*. ACM, 2018.
- [7] Seyedehzahra Khoshmanesh and Robyn R Lutz. The role of similarity in detecting feature interaction in software product lines. In *ISSREW*, pages 286–292. IEEE, 2018.
- [8] Seyedehzahra Khoshmanesh and Robyn R Lutz. Feature similarity: A method to detect unwanted feature interactions earlier in software product lines. In *International Conference on Similarity Search and Applications*, pages 356–361. Springer, 2019.
- [9] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.
- [10] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.
- [11] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *RE*, pages 139–148. IEEE, 2006.

- [12] Sven Apel, Alexander Von Rhein, Thomas Thüm, and Christian Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12), 2013.
- [13] Don Batory, Peter Höfner, and Jongwook Kim. Feature interactions, products, and composition. In *ACM SIGPLAN Notices*, volume 47, pages 13–22. ACM, 2011.
- [14] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
- [15] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *TSE*, (7), 2014.
- [16] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 482–491. IEEE Press, 2013.
- [17] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012.
- [18] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What’s in a feature: A requirements engineering perspective. In *International Conference on Fundamental Approaches to Software Engineering*, pages 16–30. Springer, 2008.
- [19] Robert J Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005.
- [20] Joanne M Atlee, Uli Fahrenberg, and Axel Legay. Measuring behaviour interactions between product-line features. In *Formal Methods in Software Engineering (FormalSE), 2015 IEEE/ACM 3rd FME Workshop on*, pages 20–25. IEEE, 2015.
- [21] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander Von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *ASE*, pages 372–375, 2011.
- [22] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 197–206. ACM, 2014.
- [23] Ana B Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 41–50. IEEE, 2014.

- [24] Muhammad Sahak, Dayang NA Jawawi, and Shahliza A Halim. An experiment of different similarity measures on test case prioritization for software product lines. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(3-4):177–185, 2017.
- [25] Paul Jaccard. É comparative study of floral distribution in a portion of the Alps and Jura. 37:547–579, 1901.
- [26] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.
- [27] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automatic and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing*. Springer Lecture Notes in Computer Science (LNCS), 2010.
- [28] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling*, pages 1–23, 2016.
- [29] Thammasak Thianniwet. *SPL-XFactor: A framework for reverse engineering feature models*. The University of Nebraska-Lincoln, 2016.
- [30] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.
- [31] Jeff Kramer, Jeff Magee, Morris Sloman, and Andrew Lister. Conic: an integrated approach to distributed computer control systems. *IEE Proceedings E (Computers and Digital Techniques)*, 130(1):1–10, 1983.
- [32] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.
- [33] M Colder and E Magilt. Feature interactions in electronic mail. *Feature Interactions in Telecommunications and Software Systems VI*, page 67, 2000.
- [34] Alan W Williams. Determination of test configurations for pair-wise interaction coverage. In *Testing of Communicating Systems*, pages 59–74. Springer, 2000.
- [35] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *International Conference on Software Product Lines*, pages 196–210. Springer, 2010.
- [36] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael 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*, pages 105–114. ACM, 2010.

- [37] Rivalino Matias, Guilherme O de Sena, Artur Andrzejak, and Kishor S Trivedi. Software aging detection based on differential analysis: an experimental study. In *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*, pages 71–77. IEEE, 2016.