IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 10, 2018, accepted September 27, 2018, date of publication October 2, 2018, date of current version October 25, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2873509

Locating Clone-and-Own Relationships in Model-Based Industrial Families of Software Products to Encourage Reuse

FRANCISCA PÉREZ, MANUEL BALLARÍN, RAÚL LAPEÑA, AND CARLOS CETINA
SVIT Research Group, Universidad San Jorge, 50830 Zaragoza, Spain

Corresponding author: Francisca Pérez (mfperez@usj.es)

This work was supported in part by the Ministry of Economy and Competitiveness through the Spanish National R+D+i Plan and ERDF funds under the project (Model-Driven Variability Extraction for Software Product Line Adoption) under Grant TIN2015-64397-R.

ABSTRACT Companies often develop similar product variants that share a high degree of functionality (i.e., features) by copying and modifying code (the clone-and-own approach). In an industrial context with a large amount of variants, software reuse can become complex for engineers. Identifying the clone-and-own relationships between the same features in different product variants can encourage reuse (e.g., suggesting improvements on how features are reused or detecting feature reuse impediments). This paper presents our approach to locate the clone-and-own relationships. To do this, our approach proposes an algorithm that combines feature location and code-comparison techniques. We evaluated our approach in three model-based industrial families of two domains (firmware for induction hobs and train control software). In our evaluation, we measure the performance (in terms of precision and recall) and compare our approach with its previous version (baseline), which uses a different technique to compare the code of each feature with its variants. The results show that our approach is able to locate clone-and-own relationships in different domains of real-world environments, and it outperforms the baseline up to 65.37% in terms of precision.

INDEX TERMS Feature location, software variability extraction, clone-and-own extraction, software maintenance and evolution.

I. INTRODUCTION

Recent research has pointed out that a family of software products inevitably contains a large amount of similar code [1] that could be reused. Companies often develop a portfolio of similar product variants which share a high degree of common functionality (i.e., features) and code, mostly due to the copy-and-paste programming practice (the *clone-and-own* approach). In an industrial context, engineers could face thousands of products that share features among them, so software maintenance and reuse could be complex.

Identifying the clone-and-own relationships across the family of products can encourage reuse. For example, clone-and-own relationships can help developers to suggest improvements on how features are reused, detect feature reuse impediments, analyze the cost-benefit payoffs of reusing code fragments against reimplementing them, and detect the maturity of a family of software products.

To encourage reuse, previous approaches have been proposed to locate features from the source code or the models of a family of products. At the code level, there are approaches [2]–[5] that isolate the implementation of the features but they do not extract the clone-and-own relationships among features. In [6], associations between artifacts are obtained by comparing the source code of existing product variants to provide hints at what features could not be separated, or for which artifacts there are multiple order options available. In [7], templates are extracted from recurring designs in source code. However, these approaches target code and do not leverage models as a source of feature location knowledge. Models have been proved to increase efficiency and effectiveness in software development [8]. Therefore, companies that develop their software products using models cannot apply these approaches to encourage reuse. At the model level, approaches target the formalization of the variability in the family of products to encourage the reuse of model fragments [9]–[12]. However, these approaches do not incorporate both feature location at model level and comparisons at code level with the goal of isolating implementations of individual features.

To cope with this lack, we propose an approach that locates the clone-and-own relationships between features in a model-based family of software products, reflecting how features are reused throughout its development. Our approach first leverages the information that existing techniques on feature location provide in order to develop an algorithm that is able to retrieve the code associated with each feature. Afterwards, our approach compares the source code of an isolated feature in a particular product against the source code of the different isolations of the same feature in other products, locating the clone-and-own relationships between the different feature isolations.

To show the feasibility and generalization of our approach, we have applied it in three industrial model-based families of software products from two domains: two model-based families of firmware for induction hobs provided by our industrial partner BSH, and a model-based family of train control PLC software provided by our industrial partner CAF. The BSH[1] group produces firmwares for their induction hobs (sold under the brands of Bosch and Siemens) over more than 15 years. CAF[2] produces PLC software to control the trains that they manufacture over more than 25 years.

The results of our evaluation show that our approach can be applied in different domains of real world environments and it is able to locate the following clone-and-own relationships between features: Reimplemented, Modified, Adapted, Unaltered, and Ghost Features. In addition, the results show that our approach outperforms the baseline in terms of precision for modified (up to 65.37%) and adapted (up to 37.5%) clone-and-own relationships, and in terms of recall for adapted (up to 48.72%) and unaltered (up to 17.95%) relationships thanks to the improved code comparison between features, which avoids that unaltered clone-and-own relationships are incorrectly classified as adapted or modified, and adapted clone-and-own relationships are incorrectly classified as modified.

This paper is an extension of a conference paper [13] and the significant differences with the conference version include: 1) The modification of the step of our approach that compares the source code of a feature in a product with the source code of the same feature in another product in order to avoid irrelevant textual differences; 2) The application of our approach in a different industrial domain (the train control PLC software provided by CAF) in order to prove its generalization; and 3) The evaluation has been further extended to measure the performance of both our approach and the baseline (the previous version of our approach) in terms of recall and precision in the three industrial case studies.

The remainder of the paper is structured as follows: Section II provides a background of the clone-and-own relationships. Section III presents our approach and shows how to apply it to a simple example. Section IV shows the evaluation of our approach in three case studies of two industrial domains. Section V summarizes the related work, and Section VI states the relevant conclusions.

[1]www.bsh-group.com
[2]www.caf.net/en

II. BACKGROUND

This section presents the different clone-and-own relationships as well as how these relationships can help developers to suggest improvements on how features are reused.

Figure 1 shows an example of a family of software products. Product A consists of two features (F1 and F2). After some time, another product (Product B) is constructed from a variant of F1 from Product A (using the clone-and-own approach), so Product B holds a clone-and-own (CAO) relationship with a previous product, Product A. Moreover, Product B comprehends a new feature (F3), which has been created from scratch. After, another product (Product C) is built with a new feature (F4), a variant of F3 from Product B, and a variant of F2 from Product A. Hence, Product C holds two clone-and-own relationships with Product A and Product B, one for each reused feature. In total, this family of products comprises 3 products, 4 features, and 3 clone-and-own relationships.

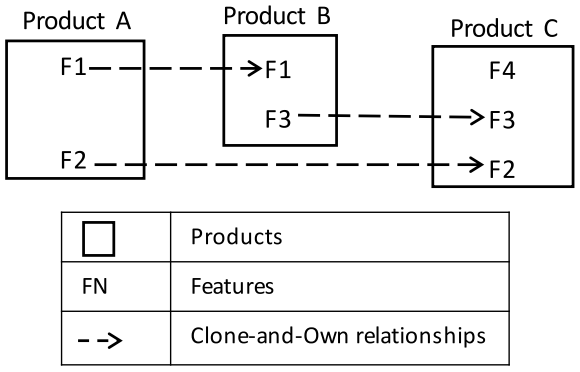

FIGURE 1. Clone-and-own relationships in a family of products.

Different clone-and-own relationships may exist in a product family. The existing relationships depend on the reuse possibilities of FN(PX) and FN(PY), where (PX) is a product that existed priorly to another product (PY), and where (FN) is a feature that is present in both (PX) and (PY) (e.g., F1(PA) and F1(PB) in Figure 1). We identified in [13] the clone-and-own relationships that Figure 2 depicts:

1) **Reimplemented Feature:** There is no shared code between FN(PX) and FN(PY). Therefore, their implementations are entirely different.
2) **Modified Feature:** There is, to some extent, code that is shared between both features. Code from FN(PX) that is also present in FN(PY) is denoted as Legacy. Differences among FN(PX) and FN(PY) are denoted as modifications.
3) **Adapted Feature:** FN(PY) includes all the code from FN(PX), plus additional novel code. Code of FN(PX) is denoted as Legacy. The novel code that causes FN(PY) and the Legacy to differ is denoted as Adapter.

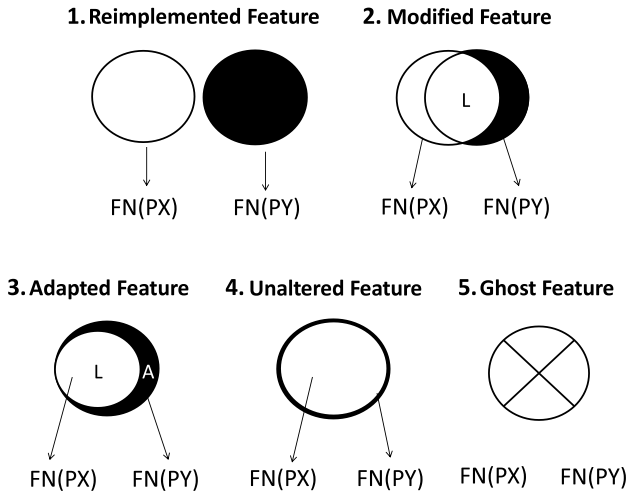

FIGURE 2. Types of clone-and-own relationships.

4) **Unaltered Feature:** The implementations of FN(PX) and FN(PY) contain the exact same code.
5) **Ghost Feature:** The FN feature is theoretically included in PY, but the approach uncovers that the code of FN is not present in PY.

The identified clone-and-own relationships may assist developers suggest improvements on feature reuse in the following ways: (1) **Reimplemented Feature** relationships help detect feature reuse barriers, indicating the existence of former implementations of features that were unrecognized by software engineers and therefore recreated from scratch, revealing missed reuse opportunities; (2) **Modified Feature** and (3) **Adapted Feature** relationships aid on the analysis of the cost-benefit trade-offs of code fragment reuse opposite to code fragment reimplementation; (4) **Unaltered Feature** relationships help detect chances to improve the reuse maturity of a software product family; and (5) **Ghost Feature** relationships highlight discrepancies between the requirements and the implementations, and therefore, the specification should be amended to refrain software engineers from wasting time trying to locate the code of those features for reuse.

For instance, Reimplemented Feature relationships may denote that a software engineer terminated his contract without transferring his knowledge of the software [13], eventually causing a fresh development of an already existing feature by another software engineer in his place. In addition to the discovery of the situation, the relationship raises awareness on both implementations, broadening the reuse possibilities. Furthermore, Unaltered Feature relationships can be utilized to assemble an implementation framework that can help in the construction of future developments.

III. THE CLONE-AND-OWN EXTRACTION APPROACH

Our approach takes as input the product models that specify a family of software products, and the product codes obtained as a result of either the translation of the models by developers or the automatic translation using a model-to-text transformation [14]. Next, our approach extracts Clone-and-Own Relationships in order to enable developers to understand and improve how features are reused among the products. Our approach builds up on feature location at the model level and code comparisons.

Figure 3 depicts the four main stages of our approach (Model-based Feature Location, Feature Isolation at model level, Feature Isolation at code level and Similarity Comparison) as well as the inputs and outputs of these stages.

We exemplify our work through the Linked List running example, based on a software products family where the variability is undefined. The products of the family have linked models, in which the code has been manually developed by a human (see left side of Figure 4). The products are either singly or doubly linked lists. Each one has a different mixture of added functionality: functionality that prints the elements of the list, functionality to sort the list through the bubble algorithm, and functionality to calculate the amount of elements of the list.

Each stage of our approach is described in the following subsections.

A. MODEL-BASED FEATURE LOCATION

In the first stage of our approach, features are extracted from the models of the products. Given a set of models, already existing Feature Location techniques can be leveraged to identify features in the models. Feature Location (identifying a fragment of source code or software model, corresponding

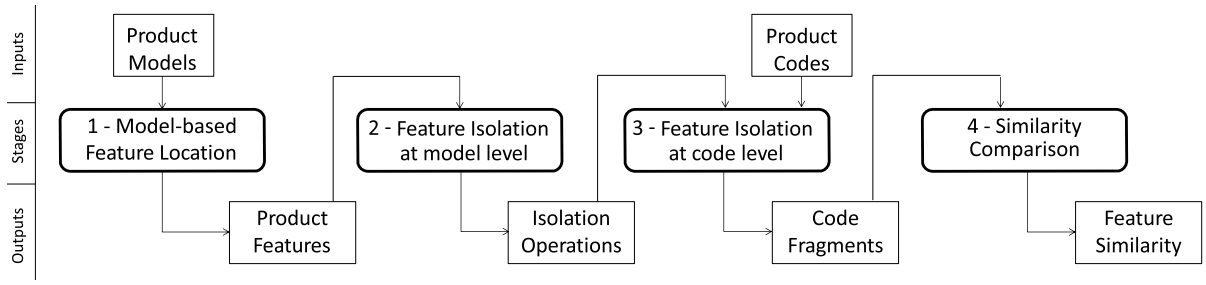

FIGURE 3. Overview of our approach.

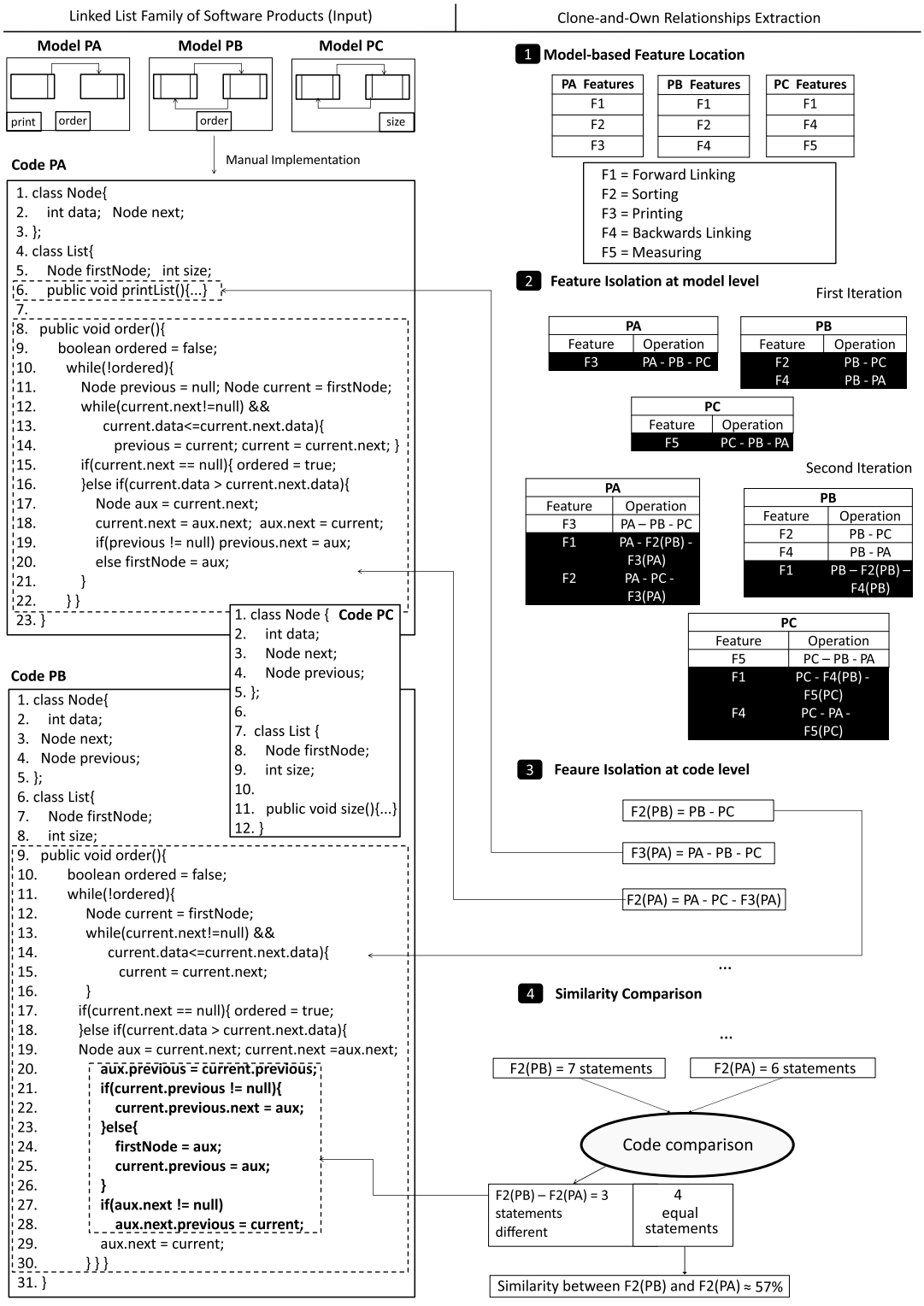

FIGURE 4. The Linked List example to show the extraction of Clone-and-own relationships.

to a specific functionality) is one of the most frequent maintenance activities undertaken by developers [15].

Several research works in literature tackle feature location in models [9], [10], [12]. For our work, we adopted Conceptualized Model Patterns to Feature Location (CMP-FL) [11]. CMP-FL identifies model patterns by human-in-the-loop (that is, through the domain knowledge of experts and engineers who participate in the process) and

then conceptualizes the extracted patterns as reusable model fragments. We adopted this technique since it allows humans to be involved in the extraction process, which improves the results since it makes that the model fragments obtained are more recognizable for humans than the model fragments obtained through automatic approaches [11].

Through CMP-FL, the elements that differ between the models are considered as alternatives for a feature, and the elements from a model that do not have a match in the rest of the models are extracted as optional features. As a result, the models are broken down into reusable fragments. Each of these reusable fragments will correspond with one of the features of the software products family. The output of the first stage of our approach is a collection of the features located in the models that belong to each product.

For example, the Linked List example of Figure 4 (see 1 in the upper-right part of the figure) tags the products (PA, PB, and PC) with the names associated with the located features (F1-F5). In the example, five features are identified within the product family. In product PA, features F1, F2, and F3 are detected. In product PB, features F1, F2, and F4 are detected. Finally, in product PC, features F1, F4, and F5 are detected.

Current techniques that locate features at the model level [9]–[12] do not provide meaningful names, only synthetic names (such as F1 or F2, for instance). We have decided to add more meaningful names to the features in order to improve the understanding of the example as the figure shows: F1 represents the Forward Linking feature, F2 represents the Sorting feature, F3 represents the Printing feature, F4 represents the Backwards Linking feature, and F5 represents the Measuring feature.

Notice that some of the features are present in more than one product: for instance, feature F2 is present in both the PA and PB products. To avoid ambiguity in the names of the features, a feature FN that belongs to a product PX will be referred to as FN(PX). For example, F2(PA) refers to F2 of PA.

B. FEATURE ISOLATION AT MODEL LEVEL

This second stage takes as input the list of the existing products and their features (which has been obtained in the previous stage) in order to perform subtractions between the different products at the model level, with the aim of isolating the code of individual features. To that extent, we developed an algorithm that determines the features that can be isolated at the model level. Each feature is accompanied by one operation, which expresses the code subtractions that need to be carried out between products to isolate the feature. The implementation of the algorithm is described through the following paragraphs.

- First of all, the algorithm creates an empty list, used to save the features that can be isolated.
- Then, the algorithm calculates the Complementary Feature Set (CFS) for each feature FN of every product PX. The CFS is a product, combination of products, or combination of products plus already isolated features, that contains all the features in PX except for FN. A CFS that contains features that are not present in PX is still valid. Subtracting the calculated CFS to PX results in the isolation of FN. With the definition of the CFS, the isolation operation is built as $FN(PX) = PX - CFS$.
- The isolated features are added to the list of isolated features along with their isolation operations. The addition of new features to the list enables for new CFS, and hence, for new feature isolation possibilities. Therefore, the algorithm performs iterations while new features are added to the isolated features list.

In the first iteration, the features that can be isolated by a CFS built through a single product or through a combination of products are included into the list. The operations found via the first iteration establish the base cases of the algorithm. In the iterations that follow, combinations between products and already calculated features serve as the CFS. The isolation operations that are found in this manner form the recursive cases of the algorithm.

Following the Linked List Example, the right side of Figure 4, part 2, shows the application of two iterations of the described algorithm:

- **First Iteration:** For all the features in PA (that is, features F1, F2, and F3), the algorithm searches for their CFS, which is only possible to calculate for F3: by removing PB and PC from PA, the code from F1, F2, F4, and F5 is eliminated from PA. Removing F1 and F2 from PA leaves us with F3, and thus, the first isolation operation is found. Notice that, while it would be enough to subtract PB from PA to achieve the same result, the criteria of eliminating the maximum possible CFS expression is followed, in order to get a purer result.

  The algorithm performs the same operation in all the products. In PB, it is possible to isolate F2 by eliminating F1 and F4 from PC, and it is also possible to isolate F4 by disposing of F1 and F2 through the removal of PA. In PC, we can isolate F5 in the same way as F3 is isolated from PA. At this point, the algorithm has gone through all the features of the family of software products, ending the iteration. As a result of the first iteration, the algorithm has finally retrieved the operations for F3(PA), F2(PB), F4(PB), and F5(PC). Since there are features that still lack an isolation operation, and since new isolation operations have been discovered in the iteration, the algorithm performs a new iteration.
- **Second Iteration:** The algorithm searches for the CFS that can isolate all the features in PA that lack an isolation operation. In order to isolate F1(PA), the algorithm removes F2 and F3 through F2(PB) and F3(PA). Moreover, F2(PA) can be isolated by subtracting PC and F3(PA) from PA.

  The same steps are followed in PB and PC. Through combinations of the different products and the features that were previously isolated, it is possible to get the

isolation operations for the features that have not been isolated yet (F1(PB), F1(PC), F4(PC)).
Therefore, the second iteration of the algorithm has produced the isolation operations for features F1(PA), F2(PA), F1(PB), F1(PC), and F4(PC). Therefore, the algorithm has isolated all the features at this point, rendering a third iteration unnecessary.

At the end, three tables are returned as output of Stage 2 (Feature Isolation at model level) of the Linked List Example (see 2 of the right part of Figure 4). Each table contains: the product name, the features that belong to it, and the isolation operations found by the algorithm.

C. FEATURE ISOLATION AT CODE LEVEL

This third stage performs the feature isolation at the code level so as to isolate the source code of the features in the products. In a software products family, novel products are implemented through increments or decrements of already existing family products. Version control software is really popular nowadays, and a wide amount of tool support for calculating differences between two source codes is available. Moreover, code comparison techniques have been used with success for large scale systems [16], [17], being the computational cost of the operation affordable should we scale up our approach. Due to all these reasons, we use diff techniques (textual comparisons) to perform code comparisons in this stage.

Following the Linked List example, features that were isolated at the model level in the second stage are now isolated at the code level. Right side of Figure 4, part 3, shows as an example the isolation of features F2(PB), F3(PA) and F2(PA). According to operation F2(PB) = PB-PC, F2(PB) can be isolated through subtracting the code of PC from PB (lines 1 to 8). Hence, the approach isolates F2 from PB (lines 9 to 30).

In order to isolate F2(PA), it is necessary to isolate F3(PA) first according to the operation F2(PA) = PA-PC-F3(PA). To do this, PB and PC are subtracted from PA according to the operation F3(PA) = PA-PB-PC. The result is the isolation of the F3 (the Printing feature from PA, declared at line 6). After, F2(PA) can be isolated by removing the code that is common between PA and PC from PA, thus removing the F3(PA) code that we just isolated. As a result, the approach isolates the F2 from PA (Sorting feature, lines 8 to 22). This stage comes to an end when the code of the features is isolated. The final output of the algorithm is the retrieved group of code fragments (one for each isolated feature).

D. SIMILARITY COMPARISON

In this last stage, the isolated code fragments that implement the features, which are in more than one product, are compared one to one in order to calculate the similarity between them. To do this, our approach performs a comparison between two fragments of code that implement the same feature.

To avoid the detection of some irrelevant textual differences in our approach, we use the technique described in [18] that computes semantic and textual differences between two programs. Although this technique does not determine precisely the set of semantic changes since it is currently limited to scalar variables, assignment statements, conditional statements, while loops, and output statements, it could detect renaming local variables as textual differences and it does not flag different extra spaces and line breaks as differences.

This technique first tries to match every component of a *New* version of a code fragment with an *Old* version that is both semantically and textually equivalent. Next, the procedure considers all unmatched components of New, attempting to match them with unmatched components of Old that are semantically equivalent but textually different. These components of New are classified as textual changes. Components of New that remain unmatched are classified as semantic changes.

Since textual changes are related to program text rather than program behavior, we only flag the semantic changes as different parts in the code. Once we obtain the equal and different parts in the code, we discard the differences and retain the equal parts of code. Feature similarity is then measured using a size metric. Size metrics are perhaps the most frequently used metrics in practice [19]. The simplest and most commonly used size metric is lines of code (LOC) but it highly depends on coding style of programmers [19]. There are other more advanced size metrics such as NIM (Number of Instance Methods) or TNOS (Total Number Of Statements). NIM counts the number of instance methods in a class, i.e., all public, protected and private methods defined in the interface of instances of a given class. The TNOS is a size metric that measures code size by counting the number of statements (e.g. for, if, return, switch, while) in each method. Since TNOS does not depend on the coding style of programmers and it is a significant predictor for the maintainability of software [19], we measure feature similarity in terms of the TNOS [15].

Following the Linked List example of Figure 4, the Similarity Comparison (part 4) compares the code of F2(PB) and F2(PA). As the code shows, the two methods are very similar, although they do not have the exact same code (the semantic changes being highlighted from lines 20 to 28 on product PB). It is reasonable for the code to differ, with PA implementing a list that is linked in a single fashion and PB implementing a list that is doubly linked. Even if the lists are sorted through the same bubble sort algorithm, said algorithm cannot be implemented in the exact same way considering the distinct number of links present between elements. From the example, it is possible to deduct that some the feature has been somehow modified since its PA implementation until its PB implementation. As a matter of fact, measuring the code, F2(PA) presents 6 statements while F2(PB) presents 7 statements. Taking in account that 4 of the 7 statements are equal, representing the same conditions in the code, the similarity percentage between F2(PA) and F2(PB) is around the 57%.

To sum up, our approach is applied to a model-based software products family with non-formalized variability. In the first stage, features from the products are identified at the model level. The second stage calculates all the possible isolation operations for the features. In the third stage, the code comparisons dictated by the calculated operations are executed to isolate the code of the features. Finally, the approach assesses the similarity degree between the features that appear in more than one product by performing a comparison in the fourth stage. The similarity between the features enables the classification of the clone-and-own relationships in one of the types described in Section II (Reimplemented, Modified, Adapted, Unaltered, or Ghost features).

IV. EVALUATION

This section presents the evaluation of our approach and the baseline, the description of the case studies where we applied the evaluation, the results obtained, the discussion, and the limitations. To evaluate the approach, we applied it to three long-living industrial case studies from two of our industrial partners: BSH, the leading manufacturer of home appliances in Europe; and CAF, an international provider of railway solutions all over the world.

A. EXPERIMENTAL SETUP

The goals of this experiment are both measuring the performance of our approach in terms of precision and recall and comparing our approach with the baseline.

Figure 5 shows an overview of the process that was followed to evaluate our approach. The left part of the figure shows the input for the evaluation process, provided by our industrial partners, which is the product family that has been specified through models. The product family is used to run our approach and the baseline. Although our industrial partners are not immune to the problem of knowledge vaporization [20], they provided us with documentation about some clone-and-own relationships and the identification of each relationship (reimplemented, modified, adapted, unaltered, or ghost). This documentation is used to build the oracle, which will be considered the ground truth and will be used to evaluate the results of our approach and the baseline.

The baseline is a previous version of our approach [13] that does not avoid the detection of irrelevant textual differences during the comparison of the source code of features (as described in Subsection III-D). We compare the clone-and-own relationships obtained in both the baseline and our approach with the oracle in order to obtain precision and recall values.

Precision measures the number of elements from the solution (*SolutionCAO*) that are correct according to the oracle (*OracleCAO*), and recall measures the number of elements of the solution (*SolutionCAO*) that are retrieved by the proposed solution (*OracleCAO*). A measure that combines both recall and precision is the harmonic mean of precision and recall, which is called the F-measure.

The recall and precision are calculated as follows:

$$Precision = \frac{SolutionCAO \cap OracleCAO}{SolutionCAO}$$

$$Recall = \frac{SolutionCAO \cap OracleCAO}{OracleCAO}$$

The F-measure that combines recall and precision is calculated as follows:

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

To calculate the precision and recall, we need to compute the true positives (TP); the number of elements in the solution that are actually correct according to the ground truth (the oracle), i.e., the clone-and-own relationships that are classified equal in both the solution and the ground truth (*SolutionCAO* $\cap$ *OracleCAO*). The precision is calculated by dividing the TP by the total number of clone-and-own relationships in the solution (*SolutionCAO*). The recall is calculated by dividing the TP by the total number of clone-and-own relationships in the oracle (*OracleCAO*). In our case, each identified clone-and-own relationship that is present in both the results and the oracle will be a TP.

Precision values can range between 0% (which means that no single clone-and-own relationship of a given type from the results is present in the oracle) to 100% (which means that all the clone-and-own relationships of a given type from the results are present in the oracle).

Recall values can range between 0% (which means that no single clone-and-own relationship of a given type obtained

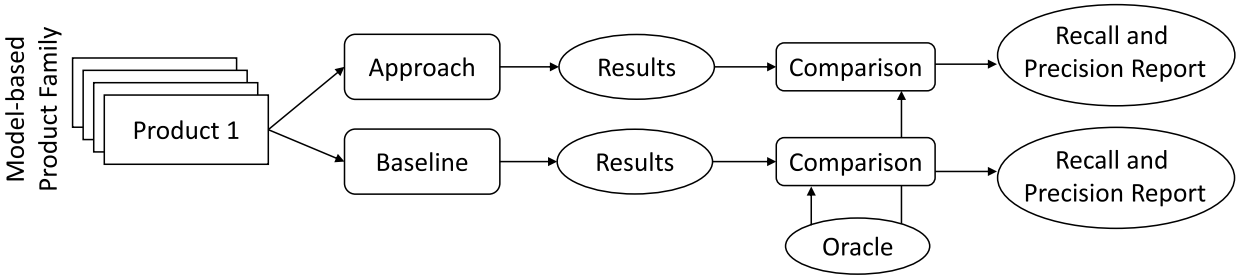

FIGURE 5. Evaluation process.

from the oracle is present in the results) to 100% (which means that all the clone-and-own relationships of a given type from the oracle are present in the results). A value of 100% precision and 100% recall implies that both identifications are the same.

1) BSH: THE INDUCTION HOBS DOMAIN

One of our industrial partners, the BSH group (`www.bsh-group.com`), has produced firmwares for their Induction Hobs (labeled under the Bosch and Siemens brands) for the last 15 years. The newest Induction Hobs (IHs) include the full cooking surface functionality, which calculates dynamic heating areas in an automatic fashion, activating or deactivating the areas depending on factors such as the utilized cookware shape, size, or position. In addition, more feedback is now provided to the user during the cooking process, including factors such as the exact cookware temperature, the temperature of the food being cooked, or real-time power consumption measurements. All of these changes have become possible through an increase of software complexity.

BSH provided us two case studies. The first case study entails a family of products that was specified using a Domain Specific Language (DSL) identified as IHDSL. After the specification, the IH's firmware was manually implemented (MI) in ANSI C by software engineers. Since this family of products belongs to BSH and uses manual implementation, we refer to this family as BSH-MI. Table 1 shows the characteristics of the BSH-MI case study. As the table shows, this family of products contains a total of 46 products and 81 features. In addition, Table 1 shows that the extracted oracle is composed by both 68 clone-and-own relationships that the features have across the different products of the family, and the identification of each clone-and-own relationship (e.g., 12 clone-and-own relationships are identified as reimplemented).

TABLE 1. Characteristics of the BSH-MI case study.

Products	46
Features	81
Total clone-and-own relationships in the oracle	68
Reimplemented	12
Modified	19
Adapted	21
Unaltered	9
Ghost	7

The second case study entails a family of products that was also specified using IHDSL. After the specification, the IH's firmware was automatically implemented (AI) using M2T (model-to-text) transformation. This transformation was produced by Acceleo [21]. Since this family of products belongs to BSH and uses automatic implementation, we refer to this family as BSH-AI. Table 2 shows the characteristics of the BSH-AI case study, which has a total of 66 products and 47 features. The oracle has 38 clone-and-own relationships in total. These relationships are identified as modified (8 relationships), adapted (11 relationships), or unaltered (19 relationships).

TABLE 2. Characteristics of the BSH-AI case study.

Products	66
Features	47
Total clone-and-own relationships in the oracle	38
Reimplemented	0
Modified	8
Adapted	11
Unaltered	19
Ghost	0

2) CAF: THE TRAIN CONTROL DOMAIN

Our other industrial partner, CAF (www.caf.net/en), has produced a family of PLC software to control the trains that they have been manufacturing over more than 25 years. Their different kinds of trains (regular trains, subway, light rail, monorail, etc.) are installed all around the globe. Train units are geared with multiple pieces of equipment through their vehicles and cabins. Equipments come from different providers that design and manufacture them with the aim of carrying out specialized tasks in the train. Some examples are the traction equipment, the brake compressors, or the power-harvesting pantograph. The train unit is also equipped with control software, which is in charge of the cooperation of the installed equipments. The control software is created with two goals in mind: (1) orchestrating the equipments to achieve flawless train functionality, and (2) guaranteeing the compliance of the train unit with the prevalent regulations of the country where the train unit is to be installed.

The DSL of CAF has enough expressiveness to describe both the interactions between the main pieces of equipment installed in a train unit and the non-functional aspects related to regulation (such as signal quality or installed redundancy levels).

An example of the functionality that the DSL can specify is the coupling between train units. A train unit can physically connect to a second train unit and control it in order to increase its passenger capacity or to rescue the second train unit in case the former suffered any damage while functioning. After the specification using the DSL, the code is obtained by means of manual implementation (MI) in C. We refer to the family of products of this case study as CAF-MI.

Table 3 shows the characteristics of the CAF-MI case study. It has a total of 23 products and 121 features, whereas the oracle has 175 clone-and-own relationships in total.

TABLE 3. Characteristics of the CAF-MI case study.

Products	23
Features	121
Total clone-and-own relationships in the oracle	175
Reimplemented	27
Modified	23
Adapted	78
Unaltered	39
Ghost	8

These relationships are identified as reimplemented (27), modified (23), adapted (78), unaltered (39), or ghost (8).

B. RESULTS

In this section, we present the results obtained for each case study in our approach and the baseline. Table 4 shows the values of precision, recall, and F-measure for each type of clone-and-own relationship (reimplemented, modified, adapted, unaltered and ghost) for the three case studies (BSH-MI, BSH-AI, and CAF-MI).

In both our approach and the baseline, the BSH-MI case study obtains the same results for precision and recall in the following relationships: reimplemented (88.89% of precision and 66.67% of recall), unaltered (87.5% of precision and 77.78% of recall) and ghost (100% of precision and 42.86% of recall). Our approach reaches better results for precision in the modified and adapted relationships, and for recall in the adapted relationship (see the shaded cells in Table 4).

In the BSH-AI case study, our approach and the baseline obtain the same results for precision and recall in the unaltered relationship, whereas our approach outperforms the baseline for precision in the modified relationship and for recall in the adapted relationship.

In the CAF-MI case study, our approach and the baseline obtain the same results for precision and recall in the reimplemented and ghost relationships. The baseline outperforms our approach for precision in the unaltered relationship, whereas our approach outperforms the baseline for precision in the modified relationship, precision and recall in the adapted relationship, and recall in the unaltered relationship.

C. DISCUSSION

The results show that there is no difference between our approach and the baseline for the reimplemented and ghost clone-and-own relationships. This is because our approach and the baseline use the same operations for the isolation of code fragments, so the code fragments used as input in the similarity step for the code comparison (which is different between our approach and the baseline) are the same in both approaches.

Reimplemented relationships are features that have been implemented from scratch, without using the source code as a template for the target code, and that have been, in some cases, developed by different teams of software engineers. Hence, the two codes that implement a reimplemented feature do not present common code. Since the two codes that implement a reimplemented feature do not share any code fragment, the result of the code comparison step is the same in both the approach and the baseline.

In case of the ghost relationships, since there is no code that implements the features, our approach and the baseline always coincide in the results.

In case of the unaltered, adapted, and modified relationships, our approach improves the results of the baseline. In both our approach and the baseline, once the source and target codes of a feature are isolated, they must be compared in

TABLE 4. Values for Precision, Recall, and F-measure in the three case studies.

		Precision		Recall		F-measure	
Case Study	Relationship	Approach	Baseline	Approach	Baseline	Approach	Baseline
BSH-MI	Reimplemented	88.89	88.89	66.67	66.67	76.19	76.19
BSH-MI	Modified	70.59	42.86	63.16	63.16	66.67	51.06
BSH-MI	Adapted	86.36	81.82	90.48	42.86	88.37	56.25
BSH-MI	Unaltered	87.50	87.50	77.78	77.78	82.35	82.35
BSH-MI	Ghost	100	100	42.86	42.86	60	60
BSH-AI	Reimplemented	-	-	-	-	-	-
BSH-AI	Modified	100	77.78	87.5	87.5	93.33	82.35
BSH-AI	Adapted	100	100	90.91	72.73	95.24	84.21
BSH-AI	Unaltered	100	100	89.47	89.47	94.44	94.44
BSH-AI	Ghost	-	-	-	-	-	-
CAF-MI	Reimplemented	72.73	72.73	88.89	88.89	80	80
CAF-MI	Modified	85.71	20.34	52.17	52.17	64.86	29.27
CAF-MI	Adapted	87.50	50	62.82	14.10	73.13	22
CAF-MI	Unaltered	66.67	69.23	41.03	23.08	50.79	34.62
CAF-MI	Ghost	100	100	75	75	85.71	85.71

order to determine which code fragments are equal between them. In case of unaltered features, incorrect code comparisons made by the baseline cause unaltered relationships to be incorrectly identified as adapted or modified relationships. In addition, the incorrect code comparisons made by the baseline cause adapted relationships to be incorrectly identified as modified. This makes the precision of the baseline worse with regard to our approach in all the case studies as follows: 27.73% for the modified relationship in the BSH manually implemented case study (BSH-MI), 22.22% for the modified relationship in the BSH automatically implemented case study (BSH-AI), and 65.37% for the modified relationship and 37.5% for the adapted relationship in the CAF manually implemented case study (CAF-MI).

Comparing the results of the automatically implemented case study with those of the two manually implemented case studies, it is possible to appreciate that the recall and precision values obtained are higher in the automatically implemented case study for both our approach and the baseline. In this scenario, the code is obtained by using a code generator, and in some cases, manually refined afterwards. Automatically generated code favors code comparisons in both our approach and the baseline because (1) the source and target code of a clone-and-own relationship that has not been manually refined should be identical and (2) when a human introduces code modifications, the refined code often uses the generated variables and methods.

D. LIMITATIONS

There are some limitations of our approach that must be acknowledged. To start with, some companies implement the code directly from requirement specifications, leading to families of software products implemented without the usage of models. Our approach, in its current state, is not applicable to said scenarios. The only stage of our approach that depends on models is the Model-based Feature Location. In order to adapt our approach to the mentioned circumstance, it would be necessary to develop techniques able to carry out feature location at the requisites level. In addition, if the features of each product are known beforehand, our approach could be adapted to start in Stage 2 (Feature Isolation).

Secondly, although in the evaluation we have chosen relationships for the oracle that can be isolated by our approach, depending on the products in the family of software products and on their particular configurations, it may not be possible for our approach to calculate all the isolation operations, or in other words, some features from some products may lack an isolation operation at the end of the execution of the algorithm. To solve this issue, our approach could suggest a selection of products with specific feature configurations designed to allow the algorithm to isolate non-isolated features. A software engineer could manually add these products to the family, enhancing the results of the algorithm.

Moreover, there is some degree of uncertainty associated with the disclosing of Clone-and-Own Relationships between products. In particular, the followed criteria is very rigid for the reimplementation and feature modification relationships. For instance, some results in reimplemented features could be incorrectly classified as modified features due to their low amounts of common code. The classification in these borderline cases is yet to be polished, and will be tackled in future works.

V. RELATED WORK

The works related to the one presented can be found in two main knowledge areas: feature location at the code level, and feature location at the model level.

A. FEATURE LOCATION AT THE CODE LEVEL

In this area, some works apply type systems to obtain relevant data when building the variability model. As an example, Typechef [22] supplies an infrastructure to analyze variability through #ifdef directives. Kästner *et al.* [23] enhance Typechef so as to support variability at run-time.

Text similarity techniques build on mathematical methods to determine textual similarity. Latent Semantic Indexing (LSI) [2] uses the number of occurrences in a set of words in large texts to obtain similarity measurements between features and source code, represented by Vector Space Models (VSM). These text similarity techniques have also been combined with dynamic analysis [3].

Other works apply reverse engineering to source code in order to obtain variability models [4], [5]. In [4], propositional logic is used to describe dependencies between features. In [24], Typechef and propositional logic are combined to extract conditions among features.

Program Dependence Analysis (PDA) is applied by several Feature Location approaches [25], [26]. PDA can be represented by Program Dependence Graphs (PDG), where nodes entail functions or global variables, and edges depict calls to functions or global variable accesses.

Trace analysis at run-time is used to define variability models through significant information. Upon execution, the technique produces traces that indicate which code has been run. Some authors [27] base their approaches on the analysis of the traces. In addition, other works mix dynamic and static analysis, such as LSI [28], PDA [27] or VSM [29].

Apart from isolating the implementations of the features, our approach also extracts Clone-and-Own Relationships among features. The relationships can be utilized by software engineers to understand in a better manner the reuse patterns of said features, and to plan and propose reuse opportunities and improvements.

Other works enhance code reuse by comparing the source code of existing product variants. In [6], associations between artifacts and their modules (i.e., features) are extracted to provide hints at what features could not be separated, or for which artifacts there are multiple order options available. In [7], recurring designs are detected in source code to extract templates as reuse opportunities. The templates can be managed and customized to generate code skeleton for the reusable features. This generated code skeleton contains

semi-implemented code that is annotated with hints and comments of necessary modifications.

In contrast to the works mentioned above that take as input source code, our approach leverages models of different product variants. When companies such as our industrial partners use models as the main software artifact to develop software in the context of the Model-Driven Development (MDD) paradigm, it is necessary to provide feedback to developers at the model level. The works mentioned above such as [6] and [7] do not consider the models, so their results are not applicable for MDD engineers. Instead, our approach considers the models, so the results are traced to the models and MDD engineers can make decisions at the model level, which is the main artifact in MDD.

B. FEATURE LOCATION AT THE MODEL LEVEL

In [9], a framework for legacy product lines mining and automated refactoring is proposed. The authors contrast the input elements, matching those with a certain degree of similarity and merging them together. The work presented in [10] proposes an approach to automatically compare products, extracting their variability in terms of the Common Variability Language (CVL) [30], [31]. Font *et al.* [11] present an approach for automating the formalization of variability in a given models family. The common and different parts of the models are specified as a set of placements over a base model and a library of model replacements. The ensuing Software Product Line (SPL) enables the derivation of new product models through the reuse of the extracted model fragments. Another approach can be found in [12], where the authors propose comparisons to extract variability from all the possible kinds of assets. All the mentioned works target the formalization of the variability inherent to an SPL. Finally, [32] identifies model patterns in a models set, conceptualizing the obtained patterns as model fragments that can be reused.

All of these approaches are limited to finding model fragments that represent features, with the ultimate goal of formalizing the variability of a particular SPL. Opposite to said works, we built an approach that incorporates both feature location at the model level and comparisons at the code level, with the goal of isolating the implementations of individual features. In addition, our work discloses Clone-and-Own Relationships among the detected features. The relationships can be used by software engineers to suggest improvements and reuse opportunities, based on the knowledge about feature reuse that the relationships expose.

VI. CONCLUSIONS

Identifying the clone-and-own relationships that are inherently present across a family of software products can help software engineers to suggest reuse improvements, to detect impediments, to analyze the cost-benefit payoffs of reuse against reimplementing, and to detect the maturity of the family.

In this paper, we have presented our approach to locate clone-and-own relationships between features in model-based families of software products. Our approach proposes an algorithm that retrieves the code associated with each feature by taking as input the information that the techniques on feature location provide. Next, our approach makes feature isolation at model and code level to obtain the source code of each feature in a particular product. Finally, our approach compares the code of the features that belong to more than one product by avoiding the detection of irrelevant textual differences in order to locate the clone-and-own relationships as Reimplemented, Modified, Adapted, Unaltered, or Ghost.

We have also shown the feasibility and generalization of our approach by applying it to real world environments of three industrial case studies in two different domains. We have successfully located the clone-and-own relationships presented in two product families of induction hob models, and in one product family of train control software. In the case of the induction hobs, one of the families had its code implemented manually and the other one, in an automatic way. In the case of the train control software, the product family had its code implemented manually.

When faced with unaltered, adapted, and modified relationships, our approach improves the results presented by the baseline. In the case of unaltered features, the baseline incorrectly classifies some of them as adapted or modified relationships. In the case of adapted relationships, the baseline sometimes makes incorrect classifications, flagging them as modified relationships. The precision of the baseline is worse than that of our approach in all the case studies: 27.73% for the modified relationship in the BSH manually implemented case study (BSH-MI), 22.22% for the modified relationship in the BSH automatically implemented case study (BSH-AI), and 65.37% for the modified relationship and 37.5% for the adapted relationship in the CAF manually implemented case study (CAF-MI).

REFERENCES

[1] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1008–1026, Sep./Oct. 2012.

[2] T. K. Landauer and J. Psotka, "Simulating text understanding for educational applications with latent semantic analysis: Introduction to LSA," *Interact. Learn. Environ.*, vol. 8, no. 2, pp. 73–86, 2000, doi: 10.1076/1049-4820(200008)8:2;1-B;FT073.

[3] F. Asadi, M. D. Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A heuristic-based approach to identify concepts in execution traces," in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering, (CSMR)*, R. Capilla, R. Ferenc, and J. C. Dueñas, Eds. Madrid, Spain: IEEE Computer Society, Mar. 2010, pp. 31–40, doi: 10.1109/CSMR.2010.17.

[4] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *Proc. 11th Int. Softw. Product Lines Conf. (SPLC)* Kyoto, Japan: IEEE Computer Society, Sep. 2007, pp. 23–34, [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.24

[5] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. Honolulu, HI, USA: ACM, May 2011, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985856

[6] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)* Washington, DC, USA: IEEE Computer Society, Sep./Oct. 2014, pp. 391–400.

[7] Y. Lin *et al.*, "Mining implicit design templates for actionable code reuse," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct./Nov. 2017, pp. 394–404.

[8] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 1st ed. San Rafael, CA, USA: Morgan Claypool Publishers, 2012.

[9] J. Rubin and M. Chechik, "Combining related products into product lines," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng. (FASE)*, in Lecture Notes in Computer Science, vol. 7212, J. de Lara and A. Zisman, Eds. Tallinn, Estonia: Springer, Mar./Apr. 2012, pp. 285–300, doi: 10.1007/978-3-642-28872-2_20.

[10] X. Zhang, O. Haugen, and B. Møller-Pedersen, "Model comparison to synthesize a model-driven software product line," in *Proc. 15th Int. Conf. Softw. Product Line Conf. (SPLC)*, E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, Eds. Munich, Germany: IEEE Comput. Soc., Aug. 2011, pp. 90–99, doi: 10.1109/SPLC.2011.24

[11] J. Font, L. Arcega, Ø. Haugen, and C. Cetina, "Building software product lines from conceptualized model patterns," in *Proc. 19th Int. Conf. Softw. Product Line (SPLC)*, New York, NY, USA, 2015, pp. 46–55. [Online]. Available: http://doi.acm.org/10.1145/2791060.2791085

[12] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proc. 19th Int. Conf. Softw. Product Line (SPLC)*, D. C. Schmidt, Ed. Nashville, TN, USA: ACM, Jul. 2015, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2791060.2791086

[13] M. Ballarín, R. Lapeña, and C. Cetina, "Leveraging feature location to extract the clone-and-own relationships of a family of software products," in *Proc. 15th Int. Conf. Softw. Reuse (ICSR)*, Limassol, Cyprus, Jun. 2016, pp. 215–230, doi: 10.1007/978-3-319-35122-3_15.

[14] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep./Oct. 2003. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231146

[15] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Softw., Evol. Process*, vol. 25, no. 1, pp. 53–95, Jan. 2013, doi: 10.1002/smr.567.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002, doi: 10.1109/TSE.2002.1019480.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006, doi: 10.1109/TSE.2006.28.

[18] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 234–245, Jun. 1990. [Online]. Available: http://doi.acm.org/10.1145/93548.93574

[19] M. Dagpinar and J. H. Jahnke, "Predicting maintainability with object-oriented metrics-an empirical comparison," in *Proc. 10th Workshop Conf. Reverse Eng. (WCRE)*, Nov. 2003, pp. 155–164.

[20] J. S. van der Ven, A. G. J. Jansen, J. A. G. Nijhuis, and J. Bosch, "Design decisions: The bridge between rationale and architecture," in *Rationale Management in Software Engineering*. Berlin, Germnay: Springer, 2006, pp. 329–348.

[21] I. Corredor, A. M. Bernardos, J. Iglesias, and J. R. Casar, "Model-driven methodology for rapid deployment of smart spaces based on resource-oriented architectures," *Sensors*, vol. 12, no. 7, pp. 9286–9335, 2012. [Online]. Available: http://www.mdpi.com/1424-8220/12/7/9286

[22] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proc. 26th Annu. ACM Conf. Object-Oriented Program. Syst. Lang. Appl.*, C. V. Lopes and K. Fisher, Eds. New York, NY, USA: ACM, 2011, pp. 805–824, [Online]. Available: http://doi.acm.org/10.1145/2048066.2048128

[23] C. Kästner, K. Ostermann, and S. Erdweg, "A variability-aware module system," in *Proc. 27th Annu. ACM Conf. Object-Oriented Program. Syst. Lang. Appl.*, G. T. Leavens and M. B. Dwyer, Eds. New York, NY, USA: ACM, Oct. 2012, pp. 773–792. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384673

[24] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. Hyderabad, India: ACM, May/Jun. 2014, pp. 140–151. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568283

[25] N. Walkinshaw, M. Roper, and M. Wood, "Feature location and extraction using landmarks and barriers," in *Proc. 23rd IEEE Int. Conf. Softw. Maintenance (ICSM)*, Paris, France, Oct. 2007, pp. 54–63. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2007.4362618

[26] M. Trifu, "Improving the dataflow-based concern identification approach," in *Proc. 13th Eur. Conf. Softw. Maintenance Reengineering (CSMR)*, A. Winter, R. Ferenc, and J. Knodel, Eds. Kaiserslautern, Germany: IEEE Computer Society, Mar. 2009, pp. 109–118, doi: 10.1109/CSMR.2009.34.

[27] A. D. Eisenberg and K. De Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance (ICSM)*. Budapest, Hungary: IEEE Computer Society, Sep. 2005, pp. 337–346, doi: 10.1109/ICSM.2005.42.

[28] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007, doi: 10.1109/TSE.2007.1016.

[29] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proc. 16th IEEE Int. Conf. Program Comprehension (ICPC)*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. Amsterdam, The Netherlands: IEEE Computer Society, Jun. 2008, pp. 53–62, doi: 10.1109/ICPC.2008.39.

[30] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Proc. 12th Int. Softw. Product Lines Conf. (SPLC)*. Limerick, Ireland: IEEE Computer Society, Sep. 2008, pp. 139–148, doi: 10.1109/SPLC.2008.25.

[31] A. Svendsen *et al.*, "Developing a software product line for train control: A case study of CVL," in *Proc. 14th Int. Conf. Softw. Product Lines (SPLC)*, in Lecture Notes in Computer Science, vol. 6287, J. Bosch and J. Lee, Eds. Jeju Island, South Korea: Springer, Sep. 2010, pp. 106–120, doi: 10.1007/978-3-642-15579-6_8.

[32] J. Font, M. Ballarín, Ø. Haugen, and C. Cetina, "Automating the variability formalization of a model family by means of common variability language," in *Proc. 19th Int. Conf. Softw. Product Line (SPLC)*, D. C. Schmidt, Ed. Nashville, TN, USA: ACM, Jul. 2015, pp. 411–418. [Online]. Available: http://doi.acm.org/10.1145/2791060.2793678

FRANCISCA PÉREZ received the Ph.D. degree in computer science from the Universitat Politècnica de València. She is currently an Assistant Professor with the SVIT Research Group, San Jorge University. Her research interests include model-driven development, variability modeling, end-user development, and collaborative modeling.

MANUEL BALLARÍN received the master's degree in computer science from San Jorge University, where he is currently pursuing the Ph.D. degree in computer science with the SVIT Research Group. He is also a Researcher with the SVIT Research Group, San Jorge University. His current research lines include model-driven development, feature location, and variability modeling.

RAÚL LAPEÑA received the master's degree in computer science from San Jorge University, where he is currently pursuing the Ph.D. degree in computer science with the SVIT Research Group. He is also a Researcher with the SVIT Research Group, San Jorge University. His main research interests lie in model-driven development, feature location, and software product lines.

CARLOS CETINA received the Ph.D. degree in computer science from the Universitat Politècnica de València. He is currently an Associate Professor with San Jorge University and the Head of the SVIT Research Group. His research focuses on software product lines, variability modeling, and model-driven development. More information about his background can be found at his website: carloscetina.com.