Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products *

Manuel Ballarin, Raúl Lapeña, Carlos Cetina

mballarin@usj.es, rlapena@usj.es, ccetina@usj.es
SVIT Research Group, San Jorge University, Zaragoza, Spain

Abstract. Feature location is concerned with identifying software artifacts associated with a program functionality (features). This paper presents a novel approach that combines feature location at the model level with code comparison at the code level to extract Clone-and-Own Relationships from a family of software products. The aim of our work is to understand the different Clone-and-Own Relationships and to take advantage of them in order to improve the way features are reused. We have evaluated our work by applying our approach to two families of software products of industrial dimensions. The code of one of the families is implemented manually by software engineers from the models that specify the software, while the code of the other family is implemented automatically by a code generation tool. The results show that our approach is able to extract relationships between features such as Reimplemented, Modificated, Adapted, Unaltered, and Ghost Features, thus providing insight into understanding the Clone-and-Own relationships of a family of software products. Furthermore, we suggest how to use these relationships to improve the way features are reused.

Keywords: Feature Location, Software Variability Extraction, Clone-and-Own Extraction

1 Introduction

Feature location is concerned with identifying software artifacts associated with a program functionality (features). Feature location is one of the most important and common activities performed by developers during software maintenance and evolution [1]. Most of the approaches carry out feature location at the code level [1],[2],[3], but in recent years feature location at the model level is gaining momentum [4],[5],[6].

This paper presents the first approach that combines the recent techniques on feature location at the model level with code comparison at the code level.

* This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO), through the Spanish National R+D+i Plan and ERDF funds under The project Model-Driven Variability Extraction for Software Product Lines Adoption (TIN2015-64397-R).

We combine both to extract Clone-and-Own Relationships from a family of software products where the software has been specified through models, and implemented either in a manual or in an automatic way. The extracted Clone-and-Own Relationships reflect how features have been reused throughout the development of the family of software products.

In order to combine both techniques, we used the information that the techniques on feature location provide to develop an algorithm that isolates features at the model level. Then, our approach uses that information to guide code comparisons at the code level. This enables us to isolate features at the code level and retrieve their source code. Finally, we make one-to-one comparisons of the source code of a feature isolated in a product with the source codes of the different isolations of the same feature in other products.

We have evaluated our approach in the industrial domain of Induction Hobs (IH) over two families of IH products. On one of them, the firmware code of the products was implemented manually from the models. On the other, the firmware code of the products was implemented in an automatic way.

The results show that it has been possible to identify several different Clone-and-Own Relationships between features such as Reimplemented, Modified,Adapted, Unaltered, and Ghost Features. These relationships are then used to suggest improvements on how features are reused. In the case of automatic implementation, extracted relationships are used to analyze whether it is necessary to carry out changes over the model-to-code transformation. In the case of manual implementation, extracted relationships are used to detect reuse impediments, to analyze cost-benefit and to detect opportunities to improve the reuse maturity.

The rest of the paper is structured as follows: Section 2 presents our approach and shows how to apply our approach to a simple example. Section 3 shows the evaluation of our work. Section 4 comprehends the work related to this paper. Section 5 summarizes the conclusions of our work.

2 Clone-and-Own Extraction Approach

The aim of our approach is to extract Clone-and-Own Relationships that enable us to understand and improve how features are reused among the products. The input of our approach is a family of software products where the software has been specified through models. The models are translated into code by humans or in an automatic way using a model-to-text transformation [7]. Our Clone-and-Own Extraction approach builds up on feature location at the model level and code comparisons. The main stages of our approach are: Model-based feature location, Feature Isolation, Code Comparison and Similarity Comparison. Fig. 1 depicts the inputs and outputs of these stages, which are described in the following subsections.

We use a running example in order to illustrate our approach. The Linked List Example is based on a family of software products where the variability is not formalized. The products have associated models, from which the code of the products has been manually implemented by a human (see left side of Fig. 2).

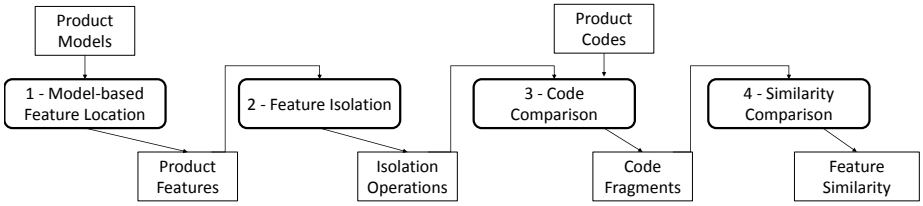

Fig. 1. Stages of the Approach

The products are lists, which can be singly or doubly linked lists. Each list has a different combination of added functionality: sorting functionality (using the bubble method), functionality that enables calculating the number of elements of the list, and functionality that prints the elements of the list.

2.1 Model-based Feature Location

The first stage of our approach extracts the features from the products at the model level by using already existing techniques that identify features given a set of models. Feature location consists of identifying a fragment in the source code or software model that corresponds to a specific functionality. It is one of the most frequent maintenance activities undertaken by developers because it is a part of the incremental change process [1].

There are several research efforts in existing literature towards feature location from a set of models [8], [6], [5]. For this stage we have adopted Conceptualized Model Patterns to feature location (hereinafter CMP-FL) [9], which identify model patterns by human-in-the-loop (domain experts and application engineers become part of the decision-making process) and conceptualize the extracted patterns as reusable model fragments. We have adopted CMP-FL because the authors show CMP-FL improves the results obtained with previous approaches, providing features that are more recognizable by the engineers.

In CMP-FL, the elements that differ between the product models are extracted as alternatives for a feature. The elements that do not have a counterpart in the rest of the models are extracted as optional features. As a result, the models will be divided into reusable model fragments. Each of the reusable fragments will correspond with one of the features of the family of software products. The output of our first stage is a list for each product, that contains the features of the product which have been located at the model level by CMP-FL.

The Linked List Example (see 1 Model-based Feature Location of Fig. 2) tags the products with the located features. In the figure, the products, their features, and the names associated with the features are shown. In this example, five features are identified in the product family.

Current techniques used to locate features at the model level [8], [6], [5] [9] do not provide meaningful names, only synthetic names (F1, F2, etc). We have decided to add more meaningful names to the features in order to improve un-

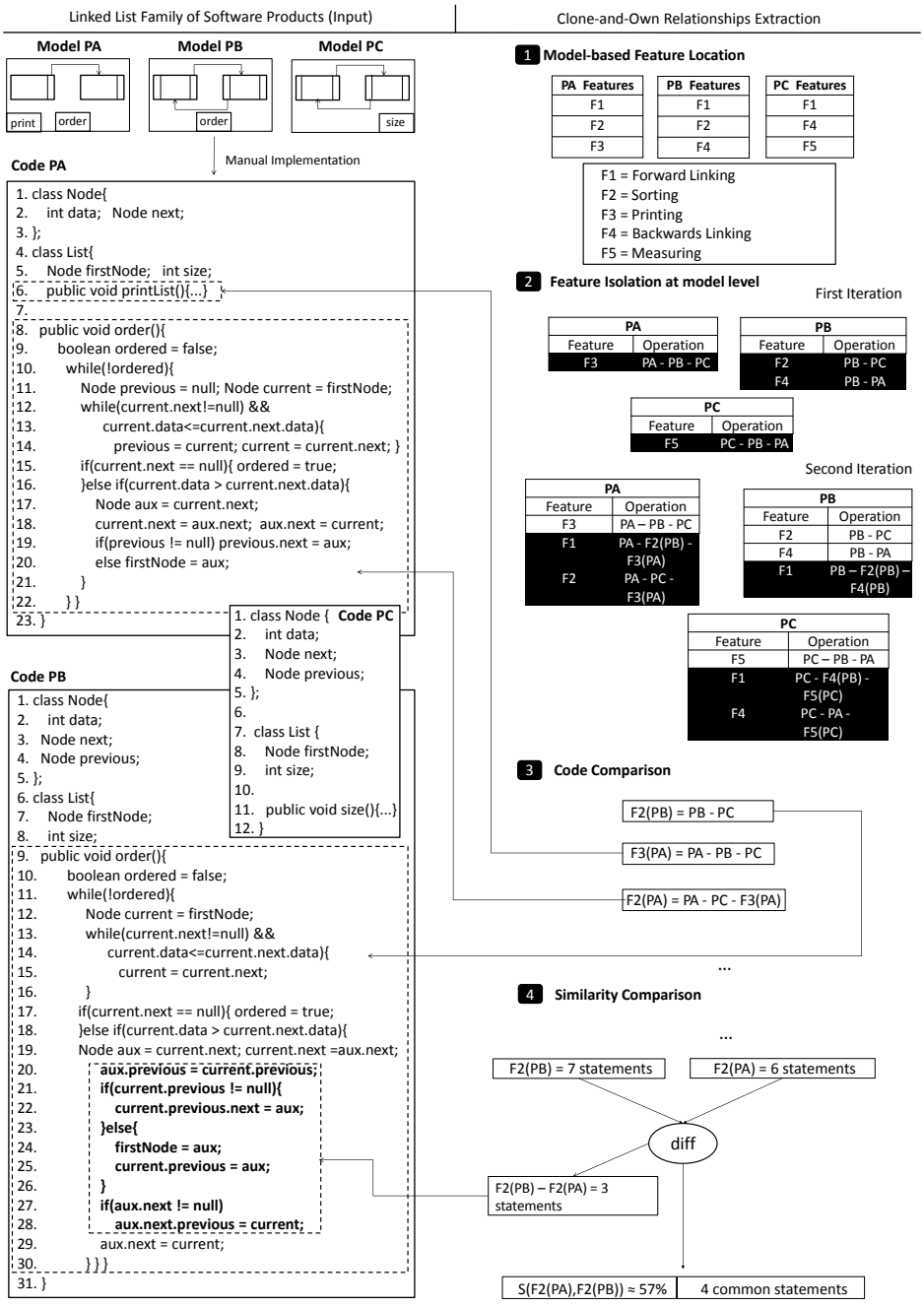

Fig. 2. Clone-and-Own Relationships Extraction applied to the Linked List Example

derstanding of the example: F1, (Forward Linking), F2 (Sorting), F3 (Printing), F4 (Backwards Linking) and F5 (Measuring).

In the first product (PA), features F1, F2 and F3 have been detected. In the second product (PB), features F1, F2, and F4 have been detected. Finally, in the third product (PC), features F1, F4, and F5 have been detected.

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

2.2 Feature Isolation

This stage performs subtractions between the different products at the model level to identify the features that can be potentially isolated in code. We developed an algorithm that performs the second stage. The algorithm's input is a list of the existing products and their features. The result of the algorithm is the list of the features that can be isolated at the model level, accompanied by one operation per feature which expresses the code subtractions that need to be carried out between products in order to isolate the mentioned feature. The implementation of the algorithm is described as follows:

- The algorithm creates an empty list to store the features that it is able to isolate.
- For each feature (FN) of every product (PX), the algorithm calculates the Complementary Feature Set (CFS). A CFS is a product, combination of products, or combination between products plus already isolated features which contains all the features in PX except for FN. A CFS is valid even if it contains features that are not present in PX. Subtracting the found CFS to PX results in isolating FN. The isolation operation becomes FN(PX) = PX - CFS (e.g.: F1(P7) = P7 - P6 - F3(P4)).
- The isolated features and their isolation operations are added to the list. The addition of new features to the list of isolated features enables for new CFS, hence new feature isolations, so we make iterations while new isolated features are added to the list.

The first iteration of the algorithm will include into the list those features that can be isolated by a CFS composed only of a product or combination of products. Isolation operations found in the first iteration constitute the base cases of our algorithm. Following iterations will use combinations between products plus already isolated features to calculate the CFS. Isolation operations found this way constitute the recursive cases of our algorithm.

The Linked List Example (see 2 Feature Isolation at model level of Fig. 2) shows the application of our feature isolation algorithm as follows.

- **First Iteration:** For all the features in PA, the feature isolation algorithm searches for the CFS that can isolate them. It is not possible to calculate

the CFS for F1 nor F2, but it is possible to calculate it for F3. Subtracting PB and PC from PA, we eliminate from PA the code from F1, F2, F4, and F5. Eliminating F1 and F2 from PA leaves us with F3. We have found the first isolation operation. Notice that it would be enough to subtract PB from PA to achieve the same result, but we follow the criteria of eliminating the maximum possible CFS expression to get a purer result.
The feature isolation algorithm performs the same search in the rest of the products. In PB, it is possible to isolate its F2 by eliminating F1 and F4 from PC, and it is also possible to isolate its F4 by disposing of F1 and F2 via PA. In PC, we can isolate F5 in a similar fashion as F3 from PA.
At this point, the feature isolation algorithm has gone through all the features of the product family, so the iteration ends. In this iteration, the feature isolation algorithm has calculated the isolation operations for F3(PA), F2(PB), F4(PB), and F5(PC). As there are still features that lack an isolation operation and we have unlocked new isolation operations, the feature isolation algorithm makes a new iteration.

– **Second Iteration:** For all the features in PA that lack an isolation operation, the feature isolation algorithm searches for the CFS that can isolate them. In order to isolate F1, we need to eliminate both F2 and F3. In the first iteration, our algorithm located F2(PB) and F3(PA). They conform the CFS for F1(PA). We can isolate F2(PA) by subtracting PC and F3(PA).
We can repeat the same steps in both PB and PC. By combining the different products and the features that we isolated in the first iteration, it is possible to get all the isolation operations for the features that lacked them in the previous step (F1(PB), F1(PC), F4(PC)).
The second iteration has calculated the isolation operations for F1(PA), F2(PA), F1(PB), F1(PC), and F4(PC). At the end of the second iteration, the feature isolation algorithm has isolated all the features, so no more iterations are needed.

As the output of the Stage 2 of the Linked List Example, three tables are returned. Each one of these tables contains the product name, the features that belong to it, and the isolation operations found by the feature isolation algorithm.

2.3 Code Comparison

The third stage runs the code comparisons specified by the operations in order to isolate the features in the source code of the products. In a family of software products, the newest products are implemented by carrying out increments or decrements of the previous products in the family. Version control software has become really popular, and there is a wide amount of tool support that calculates differences between two source codes available. Apart from this, code comparison techniques have been used successfully for large scale systems [10] [11], proving the computational cost of the operation to be affordable should we scale up our approach. For all these reasons, we use textual code comparison techniques (diff)

to execute the code comparisons dictated by the operations given by the second step of our approach.

The Linked List Example (see 3 Code Comparison of Fig. 2) shows how features are isolated. In our approach, all the features isolated at the model level in the second stage are isolated at the code level in the third stage. Due to space restrictions, this example isolates only two features: F2(PB), and F2(PA). According to the operations, F2(PB) can be automatically isolated by subtracting the code belonging to PC from PB. In this example, subtracting the code results in eliminating from PB the inner class Node and the variable declaration section (PB, lines 1 to 8). Therefore, the approach isolates the Sorting Feature from PB (PB, lines 9 to 30).

In order to isolate F2(PA), we must first isolate F3(PA). We subtract both PB and PC to PA, and after eliminating the corresponding code, the approach isolates the Printing Feature (PA, declaration at line 6). We can now isolate F2(PA) by removing from PA the code that is common between PA and PC, and disposing of the F3(PA) code that we just isolated. By doing this, the approach isolates the Sorting Feature from PA (PA, lines 8 to 22). The third stage concludes when the features are isolated in code. The output of the third stage is, for each FN(PX), the code that isolates the feature.

2.4 Similarity Comparison

In this stage, the isolated pieces of code that implement the features that belong to more than one product are compared one to one in order to calculate the similarity between them. In order to calculate the similarity between the same feature in two different products, our approach performs a diff between them.

Diff returns the equal parts and the differences in the code of the two features. We discard the code differences and retain the parts of the code that are equal between them. Similarity between features is then measured in terms of the Total Number of Statements (TNOS) [1], which is a size metric for measuring code size. TNOS counts the number of statements (e.g. for, if, return, switch, while) in each method for assessing the entire code size. This size metric is not dependent on the coding style of programmers, unlike the Lines Of Code metric.

The Linked List Example (see 4 Similarity Comparison of Fig. 2) compares F2(PB) and F2(PA). From the lines of code present in the figure, it can be appreciated that the two order methods, while very similar, do not have the exact same code (notice the marked changes from line 20 to line 28 on PB). It is reasonable, as PA implements a singly linked list and PB implements a doubly linked list. Even if the sorting technique is the same (bubble sort), it cannot be implemented the same way with a different number of links between elements. In fact, F2(PA) has 6 statements and F2(PB) has 7 statements. Considering that 4 of the 7 statements are equal and represent the same conditions in the code, the similarity percentage between F2(PA) and F2(PB) is around the 57%. From this example, we can conclude that some sort of modification has occurred to the feature since it was first implemented on PA until its appearance on PB.

Summarizing, our approach is applied to a family of software products where variability is not formalized. The first stage identifies the features from the products at a model level, tagging the products with them. Then, in the second stage, the operations to isolate the features are calculated. After that, in the third stage, the approach executes the code comparisons dictated by the operations. Finally, in the fourth stage, the approach quantifies the degree of similarity between the features that appear in more than one product. Our approach returns, for the different features in the family, the feature isolation at the code level and the degree of similarity between the features that appear in more than one product.

3 Evaluation

We have evaluated the presented ideas with our industrial partner (BSH group). Their induction division has been producing induction hobs (under the brands Bosch and Siemens among others) over the last 15 years.

3.1 The Induction Hobs Domain

The newest Induction Hobs (IHs) include full cooking surfaces, where dynamic heating areas are automatically calculated and activated or deactivated depending on the shape, size, and position of the cookware placed on top. In addition, there has been an increase in the type of feedback provided to the user while cooking, such as the exact temperature of the cookware, the temperature of the food being cooked, or even real-time measurements of the actual consumption of the IH. All of these changes are being possible at the cost of increasing the software complexity.

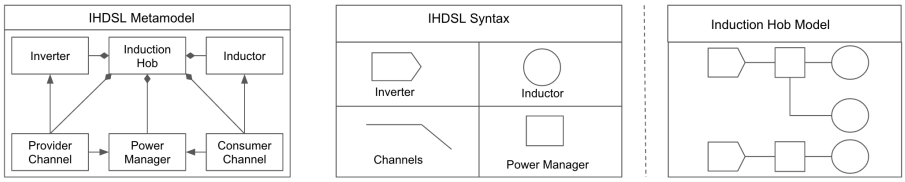

Fig. 3. IHDSL Metamodel, Syntax and Model

The Domain Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 74 references among them and more than 180 properties. However, in order to gain legibility and due to intellectual property rights concerns, in this paper we use a simplified subset of the IHDSL (see Fig. 3). The main concepts of IHDSL are: Inverter, Induction Hob, Inductor, Provider Channel, Power Manager and Consumer Channel. The firmware code of each IH is implemented in ANSI C and includes about four hundred thousand TNOS.

In order to gain legibility and due to intellectual property rights concerns, in the following lines, we explain a subset of IHDSL to present the IH domain, although in the evaluation, the complete models have been used. The main concepts of IHDSL are: Inverter, Induction Hob, Inductor, Provider Channel, Power Manager and Consumer Channel.

Inverters are in charge of converting the input electric supply to match the specific requirements of the Induction Hob. Specifically, the amplitude and frequency of the electric supply needs to be precisely modulated in order to improve the efficiency of the IH and to avoid resonance. Then, the energy is transferred to the hotplates through the channels. There can be several alternative channels, which enable different heating strategies depending on the cookware placed on top of the IH at run-time. The path followed by the energy through the channels is controlled by the power manager.

Inductors are the elements where the energy is transformed into an electromagnetic field. Inductors are composed of a conductor that is usually wound into a coil. However, inductors vary in their shape and size, resulting in different power supply needs in order to achieve performance peaks. Inductors can be organized into groups in order to heat larger cookware while sharing the user interface controllers. Each group of inductors can have different particularities; for instance, some of them can be divided into independent zones while others can grow in size adapting to the size of the cookware being placed on top of them. Some of the groups of inductors are made at design time, while others can form at run-time (depending on the cookware placed on top).

3.2 Extracted Clone-and-Own Relationships

We have applied our Clone-and-Own approach to two families of products of our industrial partner. The first family of products was specified using IHDSL. After the specification, the IH's firmware was manually implemented (MI) in ANSI C by software engineers. This family of products contains a total of 46 products. Since this family of products uses IHDSL and manual implementation we refer to this family as IHDSL+MI. The second family of products 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 [12]. This family is composed by a total of 66 products. Since this family of products uses IHDSL and automatic implementation we refer to this family as IHDSL+AI.

The IHDSL+MI family has a total of 81 different features. On the other side, the IHDSL+AI family contains a total of 47 features. After applying our Clone-and-Own Relationship extraction approach to both families of products we were able to isolate a total of 49 features belonging to IHDSL+MI and a total of 34 features belonging to IHDSL+AI. As a result, we detected five types of Clone-and-Own Relationships. Given the extracted code of FN(PX) and FN(PY), being product (PX) previous in time to product (PY), and being the same feature (FN) present in both products, we have identified the following feature relationships (see top part of Fig. 4).

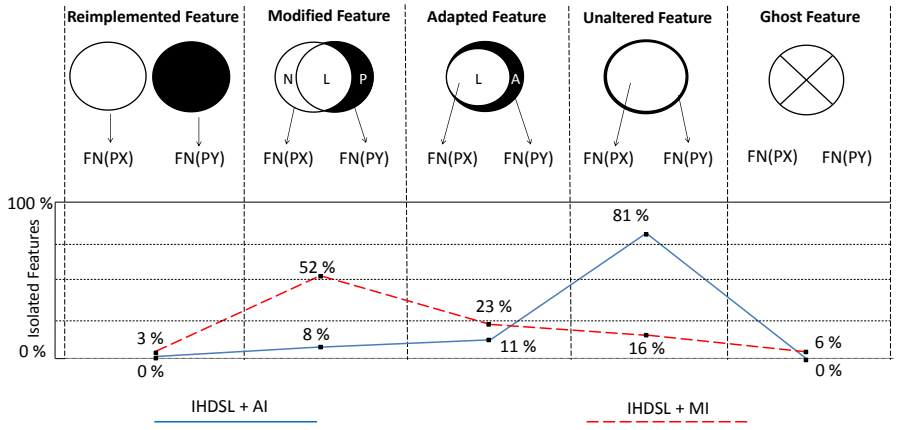

Fig. 4. Clone-and-own Relationships Extraction applied to both family of products

- **Reimplemented Feature**, FN(PX) and FN(PY) do not share code between them. The implementations of these features are entirely different.
- **Modified Feature**, it exists shared code between both features. The part of code from FN(PX) which is present in FN(PY) is referred to as Legacy. The differences between FN(PX) and the Legacy are referred to as Negative modifications. The differences between FN(PY) and the Legacy are referred to as Positive modifications.
- **Adapted Feature**, FN(PY) includes all code from FN(PX), and additional code which is not present in FN(PX). The part of FN(PX) is referred to as Legacy. Adapter represents the difference between FN(PY) and the Legacy.
- **Unaltered Feature**, the code of FN(PX) and FN(PY) is strictly the same.
- **Ghost Feature**, FN(PY) is specified at the model level but the extraction approach reveals that the code is missing.

We have the intuition that another type of relationship exists, Non-documented Features. Non-documented Features are those features that are not present at the model level, but they are at the code level. Software engineers reported that sometimes they implemented new code in later stages of the development without updating the corresponding IHDSL models. However, the full set of features of neither software family was completely isolated. The unclassified code may belong to either *Non-isolated Features* or *Non-documented Features*. Therefore, we have not evidence that this feature genuinely exists in IHDSL+MI or IHDSL+AI.

3.3 Clone-and-Own Relationships for Automatic Implementation

In the IHDSL+AI family our approach extracted the following relationships: 0% Reimplemented, 8% Modified, 11% Adapted, 81% Unaltered and 0% Ghost. The

presence of Modified and Adapted Features reveals that the implementation code of those features was refined (Modified Feature) or extended (Adapted Feature) by hand after the execution of the m2t transformation. Each feature classified as Unaltered Feature exhibits the same implementation code across all the members of the family that implement that particular feature. Unaltered Features suggest that the code of those features was not altered by software engineers after the execution of the m2t transformation.

In IHDSL+AI, the presence of Unaltered Features (81%) surpasses the presence of both Modified and Adapted Features (19%). This indicates that the m2t transformation actually saves implementation time to software engineers. Furthermore, the size of Positive modifications is smaller than the size of the Legacy feature on average (Modified Features) and the size of the Adapter is smaller than the Legacy feature on average (Adapted Features). These evidence contributes to concluding that the m2t transformation requires little human intervention.

We suggest that the Modified Feature and Adapted Feature relationships are useful to analyze whether it is necessary to carry out changes over the model-to-code transformation. If it is determined that it is necessary to update it, then the information provided by the occurrences of these relationships can be used to refine the metamodel and the code transformation rules.

In the IHDSL+AI family, modified features enabled to adjust the transformation rules. Negative parts of modified features reflected eliminated code introduced by obsolete transformation rules, and positive parts of modified features reflected manual code additions. The information provided by analyzing both the negative and positive parts enabled the company to update transformation rules with recurring changes that were predicted to keep occurring in the future.

3.4 Clone-and-Own Relationships for Manual Implementation

In the IHDSL+MI family our approach extracted the following relationships: 3% Reimplemented, 52% Modified, 23% Adapted, 16% Unaltered and 6% Ghost. The presence of Modified and Adapted Features reveals that the implementation code was reused from another product as source and then refined to meet the particularities of the target product. F2(PA) and F2(PB) of the Linked List example (see Fig. 2) are instances of the Modified Feature relationship. On one hand, both F2(PA) and F2(PB) implement the same functionality (sorting the lists using the bubble method). On the other hand the implementation details of F2(PA) are different than those of F2(PB) to accommodate a feature (F4 = Backwards Linking) of PB which is not a feature of PA.

Unaltered Features were copied from previous products and used directly in new products. It turns out, Unaltered Features are reused among different products without requiring refinements on part of the engineer to accommodate the rest of the features of the product.

In IHDSL+MI, Unaltered, Adapted and Modified Features (91%) reveal reuse opportunities identified by the software engineers. The presence of Reimplemented Features (3%) indicates that software engineers did not realize former implementations of the feature. The implementation of these features was done

from scratch, revealing missed reuse opportunities. Finally 6% of isolated features were cataloged as Ghost Features. Ghost Features reveal inconsistencies between the model specification and the implemented code. The model specification should be updated to keep software engineers from failing to locate the code of those features.

We suggest that Reimplemented Feature relationships are useful to detect feature reuse impediments. In IHDSL+MI, for instance, they were useful to detect that a developer had left the company without performing knowledge transfer, and that the new developer in his place eventually reimplemented some code from scratch. Apart from detecting the situation, now we have awareness of both implementations, therefore widening the reuse possibilities.

We propose that Modified Feature and Adapted Feature relationships are useful for analyzing cost-benefit payoffs of reusing code fragments against reimplementing them. In IHDSL+MI, for instance, 12 cases were found where it had become more costly to create adapters that allowed reusing the legacy part of a feature than to reimplement the feature as needed.

We propound that Unaltered Feature relationships are useful to detect the opportunities to improve the reuse maturity of a family of software products. In IHDSL+MI, for instance, they were useful to build an implementation framework that has been used in further developments.

3.5 Limitations

There are some limitations that must be acknowledged. To begin with, there are companies that implement the code directly from the software requirements. This leads to software product families implemented without models. In such an scenario, our approach is not applicable. Developing and using techniques that permit to carry out feature location at the requisites level would widen the scope of our approach.

Second, depending on the configuration of the products in the software family, it is possible for our feature isolation algorithm to not find the isolation operations for every feature in every product. In the future, our approach might suggest the addition of products to the family with specific feature configurations that would allow the algorithm to isolate non-isolated features.

In addition, determining the kind of Clone-and-Own Relationships between products entails some degree of uncertainty. Specifically in the cases of reimplementation and feature modification, the current criteria is very rigid. This results in reimplemented features that, due to having low amounts of common code, are incorrectly classified as modified ones.

Finally, inspecting the isolated features with domain experts, we detected that in some cases, not all the lines of code provided in an isolated piece of code belong to the isolated feature and, in some other cases, some lines that do belong to the isolated feature are missing. Nevertheless, we have confirmed that the isolated code is a good heuristic for feature location, and domain experts have validated that the behavior detected by the described Clone-and-Own Relationships is the right one at the code level.

4 Related Work

Approaches related to the one presented in this paper can be distinguished into two areas: feature location at the model level and feature location at the code level. First we introduce the state-of-the-art of feature location at the code level and secondly, the state-of-the-art of feature location at the model level.

4.1 Feature Location at the Code Level

Some works apply type systems to extract relevant information when constructing the variability model. For instance, Typechef [13] provides an infrastructure to analyze the variability with the #ifdef directives. In [14] the authors extend Typechef in order to support the variability at run-time.

Text similarity techniques are based on mathematical methods to determine the similarity in a collection of texts. As an example, Latent Semantic Indexing (LSI) [15] takes into account the number of occurrences in a set of words in large texts. LSI can be used to obtain similarity measurement metrics between features and the code used to implement them. These similarity can be represented by Vector Space Models (VSM). On some occasions text similarity techniques are combined with dynamic analysis [16].

Other works focus on applying reverse engineering to the source code to obtain the variability model [3], [17]. In [3] the authors use propositional logic which describes the dependencies between features. In [18] Typechef and propositional logic are used to extract conditions among a collection of features.

Several approaches [19], [20] apply Program Dependence Analysis (PDA) to locate features. PDA can be represented by Program Dependence Graphs (PDG) where the nodes represent functions or global variables and the edges represent function calls or accesses to global variables.

Trace analysis is a run-time technique used to define a variability model through relevant information. When the technique is executed, it produces traces indicating which parts of code have been executed. Some approaches [21] are based on traces analysis. There are also works that combine dynamic analysis and static analysis as is the case of LSI [22], PDA [21] or VSM [2].

Compared to the above works, our approach introduces software models as a new source of knowledge for feature location at the code level. Furthermore, our approach not only isolates the implemented code of the features but it also extracts Clone-and-Own Relationships among these features. These relationships are used to better understand how features are reused, and to suggest improvements on the way they are reused.

4.2 Feature Location at the Model Level

In [5], the authors propose a framework for mining legacy product lines and automating their refactoring to contemporary feature-oriented SPLE approaches. They compare the elements of the input with each other, matching those whose

similarity is above a certain threshold and merging them together. In [8], the authors propose a generic approach to automatically compare products and extract the variability among them in terms of Common Variability Language (CVL) [23], [24]. In [9] an approach to automate the formalization of variability in a given family of models is presented. The model commonalities and differences are specified as placements over a base model and replacements in a model library. The resulting Software Product Line (SPL) enables the derivation of new product models by reusing the extracted model fragments. In [6] the authors propose another approach based on comparisons to extract the variability of any kind of asset. These works focus on formalizing the variability in a SPL. Finally, [4] identifies model patterns in a set of models and conceptualizes the extracted patterns as reusable model fragments.

The above approaches limit their application to finding fragments of a model which represent features in order to formalize the variability in a SPL. In contrast, our approach combines feature location at the model level with code comparison in order to isolate the implemented code of the features. Furthermore, our work identifies several different Clone-and-Own Relationships among the located features. These relationships enable us to make improvement suggestions based on the knowledge gathered on the way features are reused.

5 Conclusions

To keep pace with the increasing demand for custom-tailored software systems, companies often apply the clone-and-own practice, through which a new product in a software product family is built by copying and adapting code from other products in the family.

In this work, we show our approach, which leverages feature location to identify and extract the Clone-and-Own Relationships from a family of software products. We have proposed an approach that extracts the features at the model level and, with that information, calculates isolation operations that enable to isolate the features at the code level. This work allows us to isolate the features of the different products in the code. With the achieved code isolation, features are compared at the code level in order to define the relationships between them.

We have evaluated the approach with our industrial partner, extracting the Clone-and-Own Relationships presented in two product families of induction hob models. One of the families had its code implemented manually and the other one, in an automatic way.

A total of five different relationships have been extracted. These relationships entitle Reimplemented, Modified, Adapted, Unaltered, and Ghost Features. The results of our approach provide insight into understanding the Clone-and-Own relationships of the features in a family of software products. These relationships are then used to suggest improvements on how features are reused.

In the case of families where automatic code generation is applied, the Modified and Adapted Features are used to analyze whether it is necessary to carry out changes over the model-to-code transformation. If it is determined that it

is necessary to improve it, then the information provided by the occurrences of these relationships can be used to refine the metamodel and the code transformation rules.

In the case of families where the code is manually implemented, Reimplemented Features are used to detect feature reuse impediments; Modified and Adapted Features are used for analyzing cost-benefit payoffs of reusing code fragments against reimplementing them; and Unaltered Features are used to detect opportunities to improve the reuse maturity of a family of software products.

References

1. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process (2013)
2. Eaddy, M., Aho, A.V., Antoniol, G., Guéhéneuc, Y.G.: CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In Krikhaar, R.L., Lämmel, R., Verhoef, C., eds.: The 16th IEEE International Conference on Program Comprehension, ICPC, Amsterdam, The Netherlands, June 10-13, 2008, IEEE Computer Society (2008) 53–62
3. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: Software Product Lines, 11th International Conference, SPLC, Kyoto, Japan, September 10-14, 2007, Proceedings, IEEE Computer Society (2007)
4. Font, J., Ballarín, M., Haugen, Ø., Cetina, C.: Automating the variability formalization of a model family by means of common variability language. In Schmidt, D.C., ed.: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville,USA, July 20-24, 2015, ACM (2015) 411–418
5. Rubin, J., Chechik, M.: Combining Related Products into Product Lines. In de Lara, J., Zisman, A., eds.: Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012 Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Volume 7212 of Lecture Notes in Computer Science., Springer (2012) 285–300
6. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Bottom-up adoption of software product lines: a generic and extensible approach. In Schmidt, D.C., ed.: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, ACM (2015)
7. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software (2003)
8. Zhang, X., Haugen, Ø., Møller-Pedersen, B.: Model Comparison to Synthesize a Model-Driven Software Product Line. In de Almeida, E.S., Kishi, T., Schwanninger, C., John, I., Schmid, K., eds.: Software Product Lines - 15th International Conference, SPLC, Munich, Germany, August 22-26, 2011, IEEE (2011) 90–99
9. Font, J., Arcega, L., Haugen, O., Cetina, C.: Building Software Product Lines from Conceptualized Model Patterns. In: Proceedings of the 19th International Conference on Software Product Line. SPLC '15, New York, NY, USA, ACM
10. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans. (2002)
11. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Trans. Software Eng. (2006)
12. Corredor, I., Bernardos, A.M., Iglesias, J., Casar, J.R.: Model-Driven Methodology for Rapid Deployment of Smart Spaces Based on Resource-Oriented Architectures. Sensors (2012)

13. Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In Lopes, C.V., Fisher, K., eds.: Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2011, ACM (2011)
14. Kästner, C., Ostermann, K., Erdweg, S.: A variability-aware module system. In Leavens, G.T., Dwyer, M.B., eds.: Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, USA, October 21-25, 2012, ACM (2012)
15. Landauer, T.K., Psotka, J.: Simulating Text Understanding for Educational Applications with Latent Semantic Analysis: Introduction to LSA. Interactive Learning Environments (2000)
16. Asadi, F., Penta, M.D., Antoniol, G., Guéhéneuc, Y.G.: A Heuristic-Based Approach to Identify Concepts in Execution Traces. In Capilla, R., Ferenc, R., Dueñas, J.C., eds.: 14th European Conference on Software Maintenance and Reengineering, CSMR, March '10, Madrid, Spain, IEEE Computer Society (2010)
17. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In Taylor, R.N., Gall, H.C., Medvidovic, N., eds.: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, ACM (2011)
18. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Mining configuration constraints: static analyses and empirical results. In Jalote, P., Briand, L.C., van der Hoek, A., eds.: 36th International Conference on Software Engineering, ICSE 14, Hyderabad, India - May 31 - June 07, 2014, ACM (2014) 140–151
19. Walkinshaw, N., Roper, M., Wood, M.: Feature Location and Extraction using Landmarks and Barriers. In: 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France, IEEE (2007)
20. Trifu, M.: Improving the Dataflow-Based Concern Identification Approach. In Winter, A., Ferenc, R., Knodel, J., eds.: 13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009, IEEE Computer Society (2009)
21. Eisenberg, A.D., Volder, K.D.: Dynamic Feature Traces: Finding Features in Unfamiliar Code. In: 21st IEEE International Conference on Software Maintenance (ICSM), 25-30 September 2005, Budapest, Hungary, IEEE Computer Society (2005) 337–346
22. Poshyvanyk, D., Guéhéneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. IEEE Trans. Software Eng. (2007)
23. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings, IEEE Computer Society (2008) 139–148
24. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K.: Developing a Software Product Line for Train Control: A Case Study of CVL. In Bosch, J., Lee, J., eds.: Software Product Lines: Going Beyond - 14th International Conference, SPLC, Jeju Island, South Korea, September 13-17, 2010. Proceedings. Volume 6287 of Lecture Notes in Computer Science., Springer (2010) 106–120