Feature Identification from the Source Code of Product Variants

Tewfik Ziadi∗, Luz Frias†, Marcos Aurélio Almeida da Silva ∗, Mikal Ziane∗
∗UMR CNRS 7606, LIP6-MoVe
4 Place Jussieu, Paris, France
†Indra Sistemas
Avenida de Bruselas, 35 Alcobendas, Madrid 28108 Spain
Email: Tewfik.Ziadi@lip6.fr, Luz.Frias@gmail.com, Marcos.Almeida@lip6.fr, Mikal.Ziane@lip6.fr

Abstract—
In order to migrate software products which are deemed similar into a product line, it is essential to identify the common features and the variations between the product variants. This can however be tedious and error-prone as it may involve browsing complex software and a lot of more or less similar variants. Fortunately, if artefacts of the product variants (source code files and/or models) are available, feature identification can be at least partially automated. In this paper, we thus propose a three-step approach to feature identification from source code of which the first two steps are automated.

Keywords—Software Product Lines; Features; reverse engineering.

I. INTRODUCTION
Software Product Lines aim at decreasing development cost and time by developing a family of systems rather than one system at a time. A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. Software Product Lines focus on capturing commonality and variability between several software products belonging to the same domain [1]. Commonality gathers assumptions that are true for all product members while variability concerns assumptions about how individual product members differ.

Software Product Line Engineering (SPLE) [1] can be implemented top-down: variability is first specified and then products are derived. The notion of features plays an important role in this top-down process [2]. Indeed, they represent a key concept to specify the variability and they are extensively used to automate the product derivation process using code refinement [3] or model specialization [4], [5]. The SPL’s top-down process is especially interesting to create new product lines. In practice however, as mentioned in [6], [7], software product lines are often set up after the implementation of several similar product variants using ad hoc reuse techniques such as copy-paste-modify.

In recent years, a lot of work [8], [9], [10], [11] has addressed the identification of features. However, most of that work takes as input textual requirements [11] or architectural artefacts [8], [9], [10]. The reverse engineering of features from the source code is seldom considered.

Assuming that an initial set of products has been chosen to include in the product line, it is then necessary to identify their features. This task can be tedious and error-prone as it may involve browsing complex software and a lot of more or less similar variants.

In this paper, we thus propose a three-step approach to feature identification from the source code of product variants.

• In the first step, a model is reverse engineered from the source code of each product. The idea is to reduce the noise induced by spurious differences in the various implementations of the same feature. Each product model is then decomposed into a set of atomic pieces.

• The second step relies on an algorithm that produces feature candidates. The algorithm identifies pieces of software that appear identical in the available products from the high-level viewpoint induced by the first step.

• The third step manually prunes the non relevant candidates that may appear for several reasons (see the discussion section) including when the level of abstraction of the model does not hide all the spurious differences in the implementations of features. This step also adds the missed features which may appear for instance when several features are always together in the considered products.

The rest of this paper is organized as follows. Section 2 introduces a simple banking example and describes our approach. Section 3 presents the results of three experiments using our approach and discusses the threats to its validity. Section 4 presents related work and the last section concludes.
II. OUR APPROACH TO FEATURE IDENTIFICATION

A. An illustrative example

As a running example, let us consider a simple banking software product line defined using a top-down approach [12]: first a feature model (see Figure 2) was manually defined. Initialization, Deposit are mandatory features, while limit, Conversion and Consortium are optional. Withdraw is an alternative feature. Second, 8 products were manually implemented using copy-paste-modify (see Figure 3 which shows capture images of the structure of their code). Each product implements a simple banking application.

The product Product1Bank for example supports limits on accounts and allows currency conversion. It consists of five classes while the Product2Bank product only includes three classes. The limit attribute and its getter and setter in the class Account are only present in four products: Product1Bank, Product4Bank, Product6Bank, and Product7Bank because they are the only products that support limit on accounts. The currency attribute and the Converter class are only present in the Product1Bank, and Product3Bank, Product3Bank and Product8Bank because they are the only products that support currency conversion.

In addition, all the products with the limit attribute are defined with the withdrawWithLimit() method in the Account class. The rest of products are defined with the withdrawWithoutLimit() method. Finally, the Consortium class is optional.

B. Product Variants Abstraction

Our approach takes as input the source code of a set of product variants but instead of analyzing the source code itself, we follow Chikofsky’s approach and rely on higher-level abstractions [13].

A structural model (a simplified UML class diagram) of each product is first extracted by reverse engineering. This assumes that products are coded using an object-oriented language. This model is then decomposed into a set of atomic pieces where each atomic piece is an elementary model construction primitive (CP). This representation is inspired by the model construction operations proposed in [14].

The construction primitives that we use to decompose the UML class diagram concerns the main elementary elements in class diagrams. This included the set of construction primitives:

$$\text{CPrimitives}_{CD} = \{ \text{CreatePackage}(\text{name}), \text{CreateClass}(\text{name, owner}), \text{CreateAttribute}(\text{name, owner}), \text{CreateOperation}(\text{name, owner}) \}$$

Each product variant $P_i$ is therefore abstracted as a set of construction primitives (SoCPs), i.e., $P_i = \{ cp_1, cp_2, ..., cp_n \}$ where each $cp_i \in \text{Primitives}_{CD}$.

Let $\text{AllIP}$ be the available set of products: $\text{AllIP} = \{ P_1, P_2, ..., P_N \}$.

At first sight this level of abstraction seems only able to capture structural differences but in object-oriented programming it is typical to reify important behavioral differences so that we expect behavioral features to be reflected in specific classes.

Figure 4 illustrates some SoCPs obtained from the banking products of Figure 4. $P_{bank}^1$, for example, is the set that represents the Product1Bank product. Note that we are not interested in primitive constructions for the model elements that represent the platform used to execute the product variant. Indeed, when reverse engineering introduces packages and classes related to the Java API (like utility and/or graphical classes), we ignore these elements to reduce the noise and only consider the elements relevant to the product domain.

C. Automatic identification of feature candidates

In order to automate the identification of feature candidates they must be formally defined. Since products are defined as sets of CPs and since feature candidates are parts of products, feature candidates will also be sets of CPs. But we do not want any set of unrelated CPs to be a feature candidate. Since we do not have access to the semantics of CPs the only information we are considering is which CPs are always present in the same products. This lead us to consider the following relation.

Definition 1 (Interdependent CPs): Given $\text{AllIP}$ a set of products, two CPs (of products of $\text{AllIP}$) $cp_1$ and $cp_2$ are interdependent iff they belong to exactly the same products $\text{AllIP}$.

- $\exists P \in \text{AllIP} \; cp_1 \in P \land cp_2 \in P$
- $\forall P \in \text{AllIP} \; cp_1 \in P \iff cp_2 \in P$

We mean feature candidates to be sets of interdependent CPs and more precisely maximal sets so that subsets of feature candidates are not considered as feature candidates. Since interdependence is obviously an equivalence relation on the set of CPs of $\text{AllIP}$ this lead us to the following definition of feature candidates.

Definition 2 (Feature candidates): A set of CPs $F$ is a feature candidate of $\text{AllIP}$ if and only if $\forall P \in \text{AllIP} \; F \subseteq P$.

1. If it is not the case, the names of functions or procedures along with the descriptions of the attributes of data structures could be used.

2. In this paper, we neither consider the parameters of operations nor the types of attributes.

3. Without semantics we cannot distinguish features that are always together in the considered products: more products, in which they are separated, will be necessary.

4. This is because in the absence of semantics, maximality helps us get rid of a lot of meaningless feature candidates.
Definition 2 (Feature candidate): Given $\text{AllP}$ a set of products, a feature candidate of $\text{AllP}$ is an equivalence class of the interdependence relation of the CPs of $\text{AllP}$.

Hence, a given set of products admits a unique set of feature candidates: the quotient set (the set of all the equivalence classes). Based on this definition, we now propose the FCIdentification algorithm to identify the feature candidates of a given set of products.

The FCIdentification algorithm takes as input a set of products deemed similar and returns the set of feature candidates of these products (this set is unique as explained previously). A sample execution of the algorithm is displayed on Figure 5.

The main data structure of our algorithm is a multiset $R$, of the occurrences of the CPs in the products of $\text{AllP}$.

Note that since all the CPs of a feature candidate appear in the same products in $\text{AllP}$, a feature candidate is included in the intersection of the products of any of its CPs. At each iteration of the loop, a feature candidate $f$ is thus built by intersecting the products of a specific CP.

To ease the definitions, $R$ is encoded as a set of couples (the disjoint union of the products) but the operators \( \setminus \) and \( \cup \) on $R$ are to be interpreted as operators on multisets of CPs.

---

**Algorithm 1 FCIdentification(AllP)**

0: Input: A set of numbered products $\text{AllP} = \{P_1, \ldots, P_n\}$, where each product of $\text{AllP}$ is a SoCP
1: Output: $F$, the set of feature candidates of the products
2: $F := \emptyset$
3: while $R \neq \emptyset$ do
4: $\text{mfcp} := \text{mostFrequentPrimitive}(R)$
5: $\text{products} := \{P_i | P_i \in \text{AllP} \land \text{mfcp} \in P_i\}$
6: $f := \bigcap_{P_i \in \text{products}} P_i \cap R$
7: $R := R \setminus f$
8: $F := F \cup \{f\}$
9: end while
10: return $F$

$\text{mfcp}$ is selected at step 4 as one of the CPs that appear most often in $R$. At step 5 the set of all the products in which $\text{mfcp}$ appears is built. At step 6 the products are intersected and then intersected with $R$ to avoid using the same CPs several times (a CP belongs to only one
feature candidate, its equivalence class). At step 7, all the occurrences of the CPs of \( f \) are removed from \( R \) and at step 8 \( f \) is added to \( F \) the set of feature candidates to be returned.

The following properties have been proven:

- The SoCPs returned by the algorithm are indeed feature candidates of \( \text{AllP} \).
- All the feature candidates of \( \text{AllP} \) are found.
- The algorithm terminates.

The application of our algorithm to the CPs of the banking product variants of Figure 3 provides the feature candidates displayed on Figure 6. The candidates have been compared to the manually identified features in Figure 2. In such a simple example all the feature candidates are in fact features so there is no need of the pruning step.

The \textbf{Base} identified feature gathers all primitive constructions related to the common elements. The feature \( F_1 \) concerns currency conversion. Indeed, it contains the primitives to create the currency attribute in the \textit{Account} class and the primitives of the \textit{Converter} class, which are necessary to exchange calculation. \( F_2 \) concerns \textit{AccountWithLimit} because it contains construction primitives to add the \textit{limit} attribute, its setter and getter and the \textit{withdrawWithLimit()} operation in the \textit{Account} class. \( F_3 \) only contains the primitives to create the \textit{withdrawWithoutLimit()} operation. This means that this feature concerns \textit{AccountWithoutLimit}. \( F_4 \) concerns the Consortium feature.

III. PRELIMINARY EVALUATION AND DISCUSSION

In this section, we present a preliminary evaluation of the approach presented in this paper and discuss the threats to its validity.

A. Preliminary evaluation

In addition to the toy banking example used in this document, our approach has been tested on two applications:

- the Graph Product Line (GPL) \cite{15} implemented using the FeatureIDE/AHEAD framework \(^6\).
- the product variants of the ArgoUML Modeling Tool \(^7\).

In the GPL experiment, 21 products were generated using the AHEAD tool. Then our approach was applied to these products. We did manage to find the original features used to produce the set of products but only

\[^6\text{http://wwwiti.cs.unimagdeburg.de/iti_db/research/featureide/}
\[^7\text{http://argouml.tigris.org/}

Figure 5. Running the \textit{FCIdentification} algorithm on a simple example.
after normalizing class names. Indeed, AHEAD generates
classes using inheritance and the name of the leaf classes
is based on the concatenation of the names of features.
The problem was that for a given feature the same classes
may be inherited in a different order among products.

In the ArgoUML experiment, the product variants that
were considered were those of the ArgoUML-SPL initially
published in [16]. We based our results on the source code
of the 10 products that is freely available for download in
the case study website\footnote{http://argouml-spl.tigris.org/}.

In the ArgoUML experiment, our prototype identified
15,100 construction primitives that define 21 candidate
features including 9 of the 10 features of original product
line (including the base feature which is common to all
the products). 10 of the 12 additional feature candidates
correspond to the implementation overhead induced by the
simultaneous presence of some features that interact with
each other. For instance, the Diagram feature impacts the
Cognitive and the Logging features. Consequently new
feature candidates are identified that represent junctions
between the original ones.

B. Discussion

The biggest advantage of our approach is that it allows
to very quickly get a set of feature candidates, thanks to
the automation of the first two steps and to the very high
level of abstraction induced by the first step.

We thus expect that for some class of legacy software
products it will prove useful even though this class still
has to be made precise by experiments on various types
of software of various complexity.

The threats to the validity of our approach are of two
kinds (which may sometimes arise for the same reasons).
First, the feature candidates may miss too many features.
Second, they may include too many irrelevant candidates.
In any case, our approach will not be useful if the third
step is too complex. The first problem may arise if some
features are always together in the considered products.
We do not expect this problem to be a major one for
several reasons. First, considering more products may be
enough to solve it. Second, splitting some features is
certainly much less time consuming and error-prone than
browsing a large source code so that the impact of the
problem is probably limited. Finally, once a split has been
decided it should be possible to add a special product
for each of the required features and then re-execute our
approach so that these features are then automatically
detected in all the products where they appear.

The first problem may also arise if the implementations
of the features are not consistent across products even at
the high level of abstraction that we considered. This
may arise if the names of classes, methods or are not uniform
across products. In this case, names must be made uniform
before applying our approach. This is what happened in
a limited way in the GPL experiment and it was possible
to get round the problem. It may also happen that a given
feature does have different classes, methods or attributes
across products, not only different naming conventions.
This suggest that the considered products may not easily
be integrated into a product line.

Another possibility which concerns both the first problem
(missed features) and the second one (too many
candidates) is that the implementation of a set of features
is not simply the union of the implementation of the
individual features (i.e. implementation is not a linear
function) to the point that feature interactions are reflected
in class, method or attribute names. This happened in the
ArgoUML experiment but at a scale that was easily dealt
with. Further research is needed to evaluate if techniques
to cope with this potential problem may be devised at least
for some classes of legacy software.

IV. RELATED WORK

There is little work on feature identification from the
source code of a set of product variants. Xue [17] proposes
some initial ideas principally based on clone detection. In-
dependently from code source, many existing approaches
consider different input artefacts of products. Alves et
al. [11] take as input textual requirements to extract a
feature model. [10], [8], [9] concern architectural artefacts.
Acher et al. [8] for example extract feature models from
plugin dependencies and architecture fragments for a set
of product variants. Yssel et al. [10] consider the extraction
of feature models from a set of similar models that represent function-blocks, a kind of architectural models for embedded systems. As discussed above, our approach can be used with this kind of artefacts that can be abstracted as CPs. For instance, it is enough to specify the set of construction primitives of function-blocks to allow identifying features from the model variants of [10].

Steve et al [18] propose an approach to the reverse engineering of feature models. They assumed, however, that the features are already identified by the user and tackle the specific problem of the construction of the feature model. In particular, they present procedures to identify alternatives from an existing set of features. Their work is complementary to our approach as it may take as input features produced by our approach.

V. CONCLUSION AND FUTURE WORK

In this paper we have proposed an approach to automate feature identification from the source code of a set of product variants. Our approach first abstracts the input products as sets of construction primitives. Then an algorithm is proposed to identify feature candidates. A third step then manually edits this set of candidates to produce the final set of features of the product line which will serve as the basis to build a feature model.

This approach has been implemented and several experiments have been conducted as a preliminary evaluation. The main advantage of our approach is that it provides a quick automatic front-end to avoid most of the tedious tasks of feature identification from source code. The experiments suggest that this approach is promising but more work is required to:

- assess the approach on more complex and more varied legacy software,
- clarify the impact of the abstractions made in the first step on the ratio of correct features identified in the second step,
- propose guidelines to the third step possibly including a first step to building a feature model.

ACKNOWLEDGMENT

This work was partly funded by the ANR Project Proteus.

REFERENCES


