JID: JSS [m5G;August 1, 2016;15:31]

The Journal of Systems and Software 000 (2016) 1–22

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Recovering software product line architecture of a family of object-oriented product variants

Anas Shatnawi[a,c,*], Abdelhak-Djamel Seriai[a], Houari Sahraoui[b]

[a] LIRMM, University of Montpellier, Montpellier, France
[b] DIRO, University of Montreal, Montreal, Canada
[c] LATECE, University of Quebec at Montreal, Montreal, Canada

A R T I C L E I N F O

Article history:
Received 17 June 2015
Revised 12 July 2016
Accepted 25 July 2016
Available online xxx

Keywords:
Software reuse
Software architecture recovery
Software product line
Object-oriented product variants
Software component
Formal concept analysis

A B S T R A C T

Software Product Line Engineering (SPLE) aims at applying a pre-planned systematic reuse of large-grained software artifacts to increase the software productivity and reduce the development cost. The idea of SPLE is to analyze the business domain of a family of products to identify the common and the variable parts between the products. However, it is common for companies to develop, in an ad-hoc manner (e.g. clone and own), a set of products that share common services and differ in terms of others. Thus, many recent research contributions are proposed to re-engineer existing product variants to a software product line. These contributions are mostly focused on managing the variability at the requirement level. Very few contributions address the variability at the architectural level despite its major importance. Starting from this observation, we propose an approach to reverse engineer the architecture of a set of product variants. Our goal is to identify the variability and dependencies among architectural-element variants. Our work relies on formal concept analysis to analyze the variability. To validate the proposed approach, we evaluated on two families of open-source product variants; Mobile Media and Health Watcher. The results of precision and recall metrics of the recovered architectural variability and dependencies are 81%, 91%, 67% and 100%, respectively.

1. Introduction

Instead of developing each software product individually, Software Product Line Engineering (SPLE) promotes a pre-planned software reuse by building and managing a family of software products that are developed in the same domain (aka. Software Product Line (SPL)) (Clements and Northrop, 2002; Pohl et al., 2005). An SPL is defined as “a set of software-intensive systems that share a common, managed set of features satisfying 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” (Clements and Northrop, 2002). The main idea behind SPLE is to analyze the business domain of an SPL in order to identify the common and the variable parts between the member products (Clements and Northrop, 2002; Pohl et al., 2005). This aims to build a software production line of a family of software products customized based on their common characteristics. Thus, a software product can be instantiated based on SPL core assets which are a set of reusable software artifacts (Clements and Northrop, 2002). SPLE is composed of two phases; domain engineering and application engineering (Pohl et al., 2005). The goal of the domain engineering phase is to create reusable core assets based on the analysis of the commonality and the variability of a family of products. Core assets consist of requirement specifications, architecture descriptions, design models, source codes, test cases, etc (Pohl et al., 2005). The goal of the application engineering phase is to (re)use core assets to derive SPL products (Pohl et al., 2005; Linden et al., 2007).

One of the most important software artifacts composing SPL's core assets is Software Product Line Architecture (SPLA) (Clements and Northrop, 2002; Linden et al., 2007; Pinzger et al., 2004). The aim of an SPLA is to highlight the commonality and the variability of an SPL at the architecture level (Pohl et al., 2005). It does not only describe the system structure at a high level of abstraction, but also describes the variability of an SPL by capturing the variability of architecture elements (Pohl et al., 2005). SPLA can be either developed from scratch, i.e. proactive strategy (Clements and Northrop, 2002; Pohl et al., 2005; Krueger, 2002), or re-engineered based on the analysis of existing software product variants, i.e extractive strategy (Krueger, 2002). However, developing SPLA from scratch is known to be a highly costly and risky task

* Corresponding author.
E-mail addresses: shatnawi@lirmm.fr, anasshatnawi@gmail.com (A. Shatnawi), seriai@lirmm.fr (A.-D. Seriai), sahraoui@iro.umontreal.ca (H. Sahraoui).

http://dx.doi.org/10.1016/j.jss.2016.07.039
0164-1212/© 2016 Elsevier Inc. All rights reserved.

(Clements and Northrop, 2002; Pohl et al., 2005; Krueger, 2002). In addition, it is common for companies to develop a set of software product variants that share common services and differ in terms of other ones. These products are usually developed in an ad-hoc manner (e.g. clone and own) by adding and/or removing some services to/from an existing software product to meet the requirement of a new need (Dubinsky et al., 2013). Nevertheless, when the number of product variants grows, managing the reuse and maintenance processes becomes a severe problem (Dubinsky et al., 2013). As a consequence, it is necessary to identify and to manage the variability between product variants as an SPL. This allows to reduce the cost of SPL development by first starting it from existing products and then being able to manage the reuse and maintenance tasks in product variants using an SPL.

In the literature, there are few approaches that recover SPLA from a set of product variants such as Frenzel et al. (2007); Koschke et al. (2009); Pinzger et al. (2004); Kang et al. (2005). These approaches suffer from two main limitations. The first one is that the architecture variability is partially addressed since they recover only some variability aspects, no one recovers the whole SPLA. The second one is that they are not fully-automatic since they rely on the expert domain knowledge which is not always available.

To address these limitations, we propose an approach to automatically recover the architecture of a set of software product variants. This is done through the exploitation of the commonality and the variability across the source code of these product variants. Our contribution is twofold: on the one hand, we recover the architecture variability concerning both component and configuration variability. On the other hand, we recover dependencies between the architectural-elements based on formal concept analysis.

In order to validate the proposed approach, we evaluated on two families of open-source product variants; Mobile Media and Health Watcher. The evaluation shows that our approach is able to identify the architectural variability and the dependencies as well. The results of precision and recall metrics of the identification of architectural variability and the dependencies are 81%, 91%, 67% and 100%, respectively.

This journal paper is an extended version of our conference paper published in Shatnawi et al. (2015b). This extension includes: (i) identifying new categories of architecture variability (e.g. internal and external component variability, variability of groups of dependencies, and dependencies related to optional component distribution). (ii) Deep analysis of the problem of SPLA identification. (iii) More details and deep analysis of the proposed solution. (iv) Related work classification. (v) Presentation of new results related to the identification of groups of variability. (vi) The pros and cons discussion. (vii) Threats to validity discussion.

The rest of this paper is organized as follows. Section 2 puts the problem in context. Next, in Section 3, we present the recovery process of SPLA. Section 4 presents the identification of architecture variability. Then, Section 5 presents the identification of dependencies among architectural-element variants. In Section 6, we identify groups of variability. Evaluation results of our approach are discussed in Section 7. A discussion about the pros and cons of our approach is placed in Section 8. Related work is analyzed in Section 9. Finally, concluding remarks and future directions are presented in Section 10.

2. Putting the problem in context

2.1. Background

2.1.1. Software product line architecture

SPLA is a special kind of software architecture. It is designed to describe the software architecture of a set of similar software products that are developed in the context of an SPL Clements and Northrop (2002). In the literature, many definitions have been presented to define SPLA. These definitions consider SPLA as a core architecture that captures the variability of a set of software products at the architecture level. However, they differ in terms of the variability definition. For instance, DeBaud et al. (1998) defined an SPLA as an architecture shared by their member products and has such a variability degree. This is a very general definition since it does not specify the nature of the architecture variability. In contrast, Pohl et al. (2005) provide a more accurate definition by specifying the nature of architecture variability. In this definition, SPLA includes variation points and variants that are presented in such a variability model. Gomaa (2005) links the architecture variability with the architectural-elements. Thus, in his definition, SPLA defines the variability in terms of mandatory, optional, and variable components, and their connections.

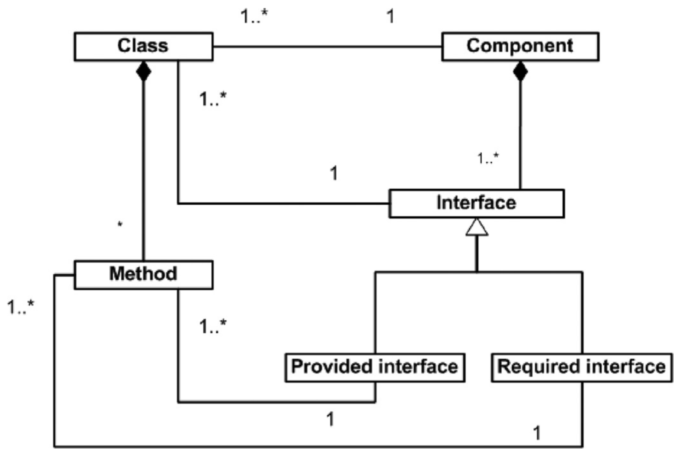


Fig. 1. UML object-to-component mapping model.

2.1.2. Component-based architecture recovery from single software: the ROMANTIC approach

For the evaluation, we use our previous works (Kebir et al., 2012b; Chardigny et al., 2008a), the ROMANTIC[1] approach, to automatically recover a component-based architecture from the source code of a single object-oriented software. Components are obtained by partitioning classes constituting the implementation of this software. Each class is assigned to a unique subset forming the implementation of an object-oriented component, i.e. a component that can be implemented using an object-oriented component model such as OSGi (Tavares and Valente, 2008). ROMANTIC is based on two main models. The first concerns the object-to-component mapping model which allows to link object-oriented concepts (e.g. package, class) to component-based ones (e.g. component, interface). Following this model, a component consists of two parts; internal and external structures. The internal structure is implemented by a set of classes that have direct links only to classes that belong to the component itself. The external structure is implemented by the set of classes that have direct links to other components' classes. Classes that form the external structure of a component define the component interface. Fig. 1 shows the object-to-component mapping model. The second model proposed is used to evaluate the quality of the recovered architectures and their architectural-element. For example, the quality-model of recovered components is based on three characteristics; composability, autonomy and specificity. These refer respectively to the

[1] ROMANTIC: Re-engineering of Object-oriented systeMs by Architecture extractioN and migraTIon to Component based ones.

Table 1
Formal context example.

	Natural	Artificial	Stagnant	Running	Inland	Maritime	Constant
River	X			X	X		X
Sea	X		X			X	X
Reservoir		X	X		X		X
Channel				X	X		X
Lake	X		X		X		X

ability of the component to be composed without any modification, to the possibility to reuse the component in an autonomous way, and to the fact that the component implements a limited number of closed services. Based on these models, ROMANTIC defines a fitness function applied in a hierarchical clustering algorithm (Kebir et al., 2012b; Chardigny et al., 2008a) as well as in search-based algorithms (Chardigny et al., 2008b) to partition the object-oriented classes into groups, where each group represents a component.

2.1.3. Formal concept analysis

To analyze the architectural variability, we use Formal Concept Analysis (FCA). It is a mathematical data analysis technique developed based on the lattice theory (Ganter and Wille, 1996). It allows the analysis of the relationships between a set of objects described by a set of attributes. In this context, maximal groups of objects sharing the same attributes are called formal concepts. These are extracted and then hierarchically organized into a graph called a concept lattice. Each formal concept consists of two parts. The first allows the representation of the objects covered by the concepts called the extent of the concept. The second allows the representation of the set of attributes shared by the objects belonging to the extent. This is called the intent of the concept. Concepts can be linked through sub-concept and super-concept relationships (Ganter and Wille, 1996) where the lattice defines a partially ordered structure. A concept A is a sub-concept of the super-concept B, if the extent of the concept B includes the extent of the concept A and the intent of the concept A includes the intent of the concept B.

The input of FCA is called a formal context. A formal context is defined as a triple $K = (O, A, R)$ where O refers to a set of objects, A refers to a set of attributes and R is a binary relation between objects and attributes. This binary relation indicates a set of attributes that are held by each object (i.e. $R \subseteq OXA$). Table 1 shows an example of a formal context for a set of bodies of water and their attributes. An X refers to the fact that an object holds an attribute.

As stated before, a formal concept consists of extent E and intent I, where E a subset of objects O ($E \subseteq O$) and I a subset of attributes A ($I \subseteq A$). A pair of extent and intent (E, I) is considered a formal concept, if and only, if E consists of only objects that share all attributes in I and I consists of only attributes that are shared by all objects in E. The pair (”river, lake”, ”inland, natural, constant”) is an example of a formal concept of the formal context in Table 1. Fig. 2 shows the concept lattice of the formal context presented in Table 1.

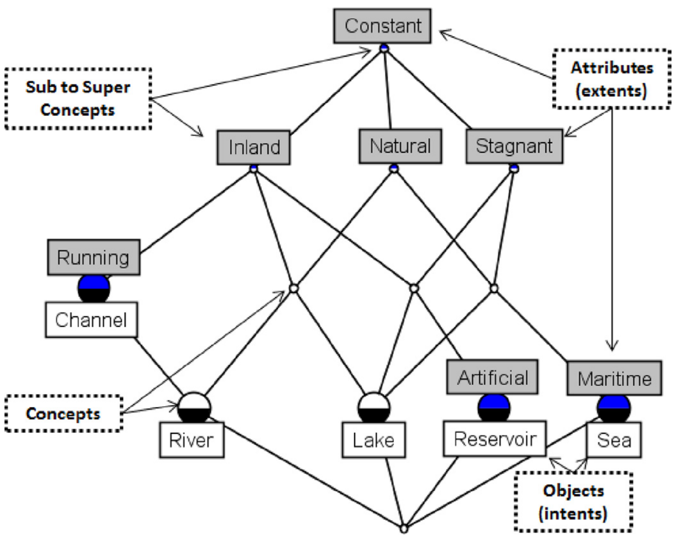


Fig. 2. The concept lattice of the formal context in Table 1.

2.2. Problem analysis

Software variability is the main theme of SPLE. It is related to the susceptibility and flexibility of software to change (Clements and Northrop, 2002). The variability in an SPL is realized at different levels of abstraction during the development life cycle, e.g. requirement, and design. For instance, at the requirement level, it is originated starting from the differences in users' wishes, and does not carry any technical sense (Pohl et al., 2005). This is related to a set of features that are needed to be included in such an application (e.g. the user needs camera, WIFI, and color screen features in the phone). Usually, this variability is documented by feature modeling language (Kang et al., 1990). At the design level, the variability has more details related to technical solutions to form the application architectures. These technical details describe how the applications are built and implemented with regard to the point of view of software architects (Pohl et al., 2005). Such technical details are those related to which software components are included in the application (e.g. video recorder, photo capture, and media store components), how these components interact through their interfaces (e.g. video recorder provides a video stream interface to media store), and what topology forms the architectural configuration (i.e. how components are composited) (Nakagawa et al., 2011). All of these technical details are described via Software Product Line Architecture (SPLA) (Pohl et al., 2005).

SPLA realizes the software variability at the architecture level by exploring the commonality and the variability of architecture elements, i.e. component, connector and configuration variability. In this paper, we focus on component and configuration variability. We do not consider connector variability since the connectors are not considered as first class concepts in the most of architecture description languages such as Magee and Kramer (1996) Luckham (1996) Canal et al. (1999). To better understand the architecture variability, we rely on the example provided in Fig. 3. This example schemes the architecture of three product variants related to an audio player product family. Each architecture variant diverges in the set of components constituting its architecture as well as the links between these components.

Components are considered as the main building unit of an architecture. Their variability can be considered following two dimensions. The first one is related to the existence of several components having the same architectural meaning, i.e. almost provide

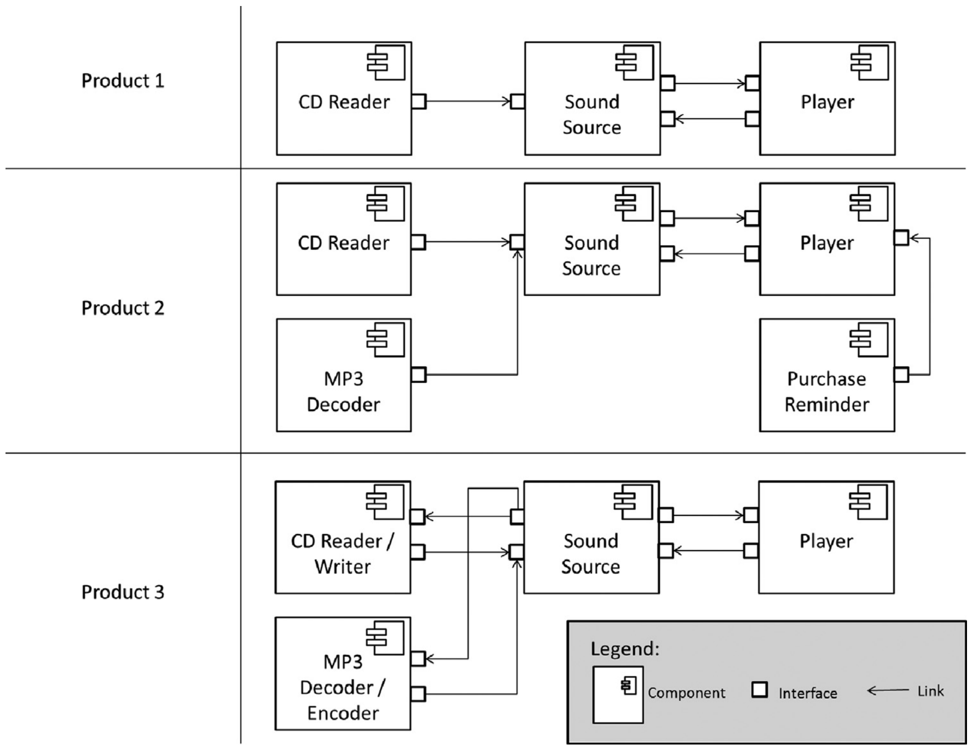


Fig. 3. An illustrative example of architecture variability.

the same services. We call these component variants. For example, in Fig. 3, *MP3 Decoder* and *MP3 Decoder/Encoder* are examples of component variants. The second dimension is related to the commonality and the variability between component variants. This is realized though internal and external variability. Internal variability refers to the divergence related to the implementation details of component variants which may lead to variability in the set of services provided by these component variants, e.g. *Decoder* only or *Decoder/Encoder* services. External variability refers to the way that the component interacts with other components. This is realized through the variability in the component interfaces. In our example, the *Sound Source* component variants have either one or two interfaces.

Furthermore, the architecture configuration does not only define the topology of how components are composited and connected, but also defines the set of included components. Thus, the configuration variability is represented in terms of presence/absence of components, on the one hand, and presence/absence of component-to-component-links on the other hand. These respectively refer to the commonality (mandatory) and the variability (optional) of components and component-links. For example, in Fig. 3, *Sound Source* is a mandatory component, while *Purchase Reminder* is an optional one. The links that connect *Player* and *Sound Source* are mandatory links, while the link that connects *MP3 Decoder/Encoder* and *Sound Source* is an optional one.

The identification of components and component-links variability is not enough to define a valid architectural configuration. It also depends on the identification of architectural-element dependencies, i.e. constraints, that may exist between the elements of the architectures. For instance, components providing antagonism services have an exclude dependency. Furthermore, a component may need other components to perform its services. This refers to a required dependency.

3. Architecture variability recovery process

The goal of our approach is to recover SPLA of a set of product variants. This is obtained by identifying variability among architectures respectively recovered from each single product. Thus, we identify component-based architecture by analyzing the object-oriented source code of each single product. This constitutes the first step of the recovery process. To identify architectural variability among the identified component-based architectures, we identify component variants based on the identification of components providing similar functionalities. This is the role of the second step of the recovery process. Next, we identify configuration variability based on both the identification of mandatory and optional components as well as links between these components. In addition, we capture the dependencies among optional components based on FCA. These are mined in the fourth step of the recovery process. Fig. 4 shows these steps.

4. Identifying the architecture variability

The architecture variability is materialized either through the existence of variants of the same architectural-element (i.e. component variants) or through the configuration variability. In this section, we show how component variants and configuration variability are identified.

4.1. Identifying component variants

The selection of a component to be used in an architecture is based on its provided and required services. The provided services define the role of the component. However, other components may provide the same, or at least similar, core services. Each component may also provide other specific services in addition to the core ones. Considering these components, either as completely different or as the same, does not allow the variability related to components to be captured. Thus, we consider them as

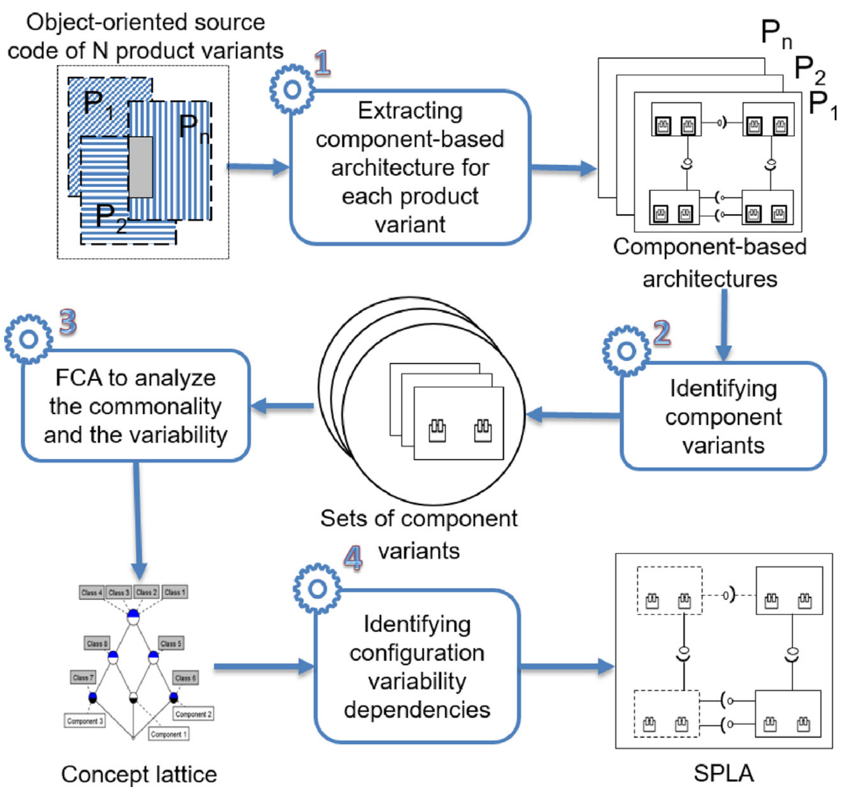

Object-oriented source code of N product variants
P1
Pn
P2
1
Extracting component-based architecture for each product variant
Pn
P2
P1
Component-based architectures
2
Identifying component variants
Sets of component variants
3
FCA to analyze the commonality and the variability
Concept lattice
4
Identifying configuration variability dependencies
SPLA

Fig. 4. The process of architectural variability recovery.

component variants. We define component variants as a set of components providing the same core services and differing concerning a few secondary ones. In Fig. 3, *MP3 Decoder* and *MP3 Decoder/Encoder* are considered as component variants. In this section, we identify component variants based on the identification of components providing similar services. Next, we analyze their variability in terms of internal and external variability.

4.1.1. *Identification of components providing similar services*

We identify component variants based on their similarity. Similar components are those sharing the majority of their classes and differing in relation to some others. The identification of similar components is based on the strength of similarity realized in their implementing classes. To do this, we use cosine similarity metric (Han et al., 2006) where each component is considered as a text document composed of the names of its classes. We use a hierarchical clustering algorithm (Han et al., 2006) to gather similar components into clusters. It starts by considering components as initial leaf nodes in a binary tree. Next, the two most similar nodes are grouped into a new one that forms their parent. This grouping process is repeated until all nodes are grouped into a binary tree (see Algorithm 1). Note that two components from the same product can not be grouped together in the same cluster. The clustering algorithm checks this situation and forbids it All nodes in this binary tree are considered as candidates to be selected as groups of similar components. To identify the best nodes, we use a depth first search algorithm (see Algorithm 2). Starting from the tree root to find the cut-off points, we compare the similarity of the current node with its children. If the current node has a similarity value exceeding the average similarity value of its children, then the cut-off point is in the current node. Otherwise, the algorithm continues through its children. The results of this algorithm are clusters where each one is composed of a set of similar components that represent variants of one component.

Algorithm 1: Building dendrogram for similar component identification.

```
Input: Components(PC)
Output: Dendrogram Tree (dendrogram)
BinaryTree dendrogram = PC;
while (|dendrogram| > 1) do
    c1, c2 = mostLexicallySimilarNodes(dendrogram);
    c = newNode(c1, c2);
    remove(c1, dendrogram);
    remove(c2, dendrogram);
    add(c, dendrogram);
end
return dendrogram
```

Algorithm 2: Dendrogram traversal for similar component identification.

```
Input: Dendrogram Tree(dendrogram)
Output: A Set of Clusters of Components(clusters)
Stack traversal;
traversal.push(dendrogram.getRoot());
while (! traversal.isEmpty()) do
    Node father = traversal.pop();
    Node left = dendrogram.getLeftSon(father);
    Node right = dendrogram.getRightSon(father);
    if similarity(father) > (similarity(left) + similarity(right) / 2) then
        clusters.add(father)
    else
        traversal.push(left);
        traversal.push(right);
    end
end
return clusters
```

Table 2
An example of formal context of three component variants.

	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8
Variant 1	X	X	X	X	X			X
Variant 2	X	X	X	X	X	X		
Variant 3	X	X	X	X			X	X

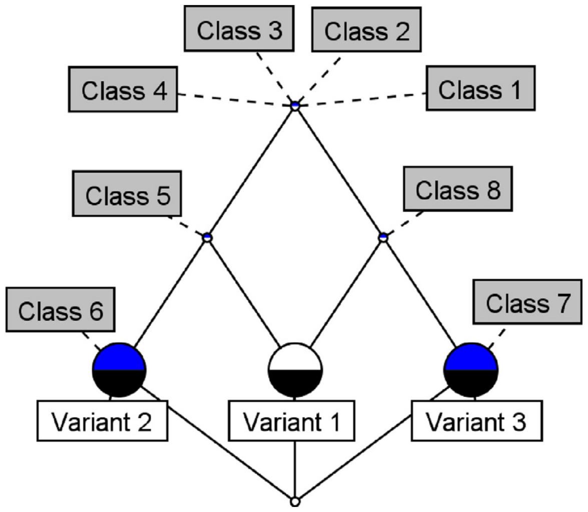


Fig. 5. A lattice example of component variants.

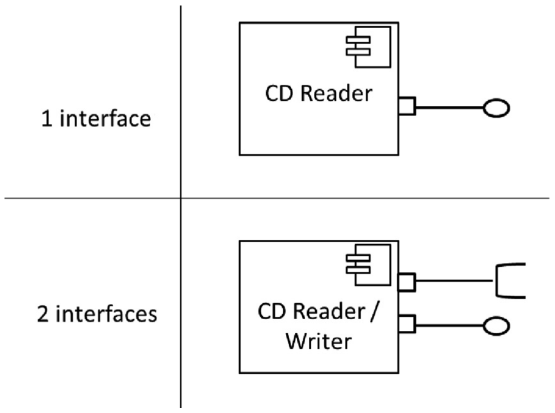


Fig. 6. An example of interface variability.

4.1.2. Identification of internal variability

Internal structure variability is related to the implementation of components. Component variants are implemented by object-oriented classes. Thus, we identify internal variability in terms of class variability. We distinguish two categories of classes. The first one refers to classes related to commonality. These belong to all variants of the component. We call them common classes. The second category refers to classes related to variability. These do not belong to all variants of the component. We call them variable classes.

The identification of common and variable classes and the distribution of variable classes is achieved using Formal Concept Analysis (FCA). To this end, we build the formal context, so that each component variant is considered as an object and each class is an attribute in this formal context. Table 2 shows an example of a formal context built using three component variants. A cross in the cell (*V, C*) denotes that the variant *V* holds the class *C*. In the lattice generated based on this formal context, common classes are grouped in the root, while the variable ones are hierarchically distributed to the non-root nodes. The leaf nodes represent component variants. These variants have all classes that are attached to the nodes placed in the path to the root node.

Fig. 5 shows the lattice extracted based on the formal context presented in Table 2. On the one hand, the commonality of their implementation is represented by common classes grouped together on the top of the lattice (i.e. the root). *Class 1, Class 2, Class 3* and *Class 4* are the common classes. On the other hand, the variability of their implementation is represented by variable classes distributed on the non root nodes. For instance, *Class 8* belongs to two variants; *Variant 1* and *Variant 3*, while *Class 6* belongs only to one variant; *Variant 2*.

4.1.3. Identification of external variability

The interaction between components is realized through their interfaces; provided and required interfaces. Provided interfaces are abstract descriptions of services provided by a component. These services may be required by other components in the same architecture. In object-oriented components, an interface is the abstraction of a group of method invocations. This group provides an access to the component services. As component variants are identified from different products, thus each variant may have some interfaces that are different compared to the other variants. For example, in Fig. 6, *CD Reader* and *CD Reader/Writer* variants differ compared to their interfaces. In the former, it has only one provided interface that provides the service of reading the *CD* content. In the latter, it has an additional required interface compared to the first variant. This interface requires a source of information to be written on the *CD*. The interfaces of a component can be classified into mandatory and optional interfaces. Mandatory interfaces are ones existing in all variants of the component (e.g. the provided interface in Fig. 6). Optional interfaces are those that are not mandatory (e.g. the required interface in Fig. 6).

Algorithm 3 shows the procedure for identifying mandatory and optional interfaces of a set of component variants. For each variant, we identify its interfaces in the corresponding products in which the variant has been identified. This is done using our approach presented in Kebir et al. (2012b) which identifies interfaces as groups of methods. To identify whether interfaces are similar or not, we rely on the textual similarity between the source

Algorithm 3: Identifying interface variability.

```
Input: A Set of Component Variants ($CV$)
Output: A Set of Mandatory and Optional Interfaces ($MI$, $OI$))
$MI$ = identifyInterfaces($CV$.getFirstVariant());
$allInterfaces = \emptyset$;
$Identify the union and intersection of component variant interfaces$
for each $v \in CV$ do
    $MI = MI \cap$ identifyInterfaces($v$);
    $allInterfaces = allInterfaces \cup$ identifyInterfaces($v$);
end
$OI = allInterfaces - MC$;
return $MI, OI$
```

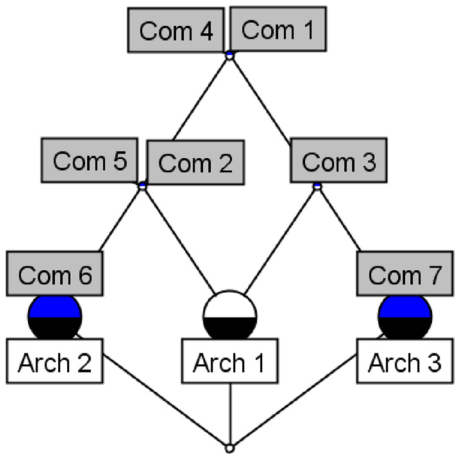


Fig. 7. A lattice example of similar configurations.

code of their implemented methods. In this context, we compare the complete method implementations. If the similarity exceeds a pre-defined threshold value defined by the software architects based on their knowledge about the variability degree of product variants, then they are considered as the same interface. Software architects can use software metrics presented in Berger et al. (2010) that measure the variability degree of a product variants based on commonality size, impact of commonality, etc. The intersection of the sets of interfaces from all the products determines all mandatory interfaces for the given component. Interfaces that do not belong to this intersection are optional ones.

4.2. Identifying configuration variants

In the previous section, we identified component variability in terms of component variants and their internal and external variability. In this section, we present how to recover the configuration variability. The architectural configuration is defined based on the list of components composing the architecture, as well as the topology of the links existing between these components. Thus, the configuration variability is related to these two aspects; the lists of mandatory (core) and optional components and the list of mandatory and optional links between the selected components.

4.2.1. Identification of component variability

To identify mandatory and optional components, we use Formal Concept Analysis (FCA) to analyze architecture configurations. We present each software architecture as an object and each member component as an attribute in the formal context. In the concept lattice, common attributes are grouped into the root while the variable ones are hierarchically distributed among the non-root concepts.

Fig. 7 shows an example of a lattice for three similar architecture configurations. The mandatory components are grouped together at the root concept of the lattice (the top). In Fig. 7, *Com 1* and *Com 4* are the mandatory components presented in the three architectures. By contrast, optional components are represented in all lattice concepts except the root. e.g., according to the lattice of Fig. 7, *Com 2* and *Com 5* are presented in *Arch 1* and *Arch 2* but not in *Arch 3*.

4.2.2. Identification of component-link variability

A component-link is defined as a connection that materializes the composition of two components respectively through their provided and required interfaces. Fig. 8 shows an example of how components are linked through their interfaces.

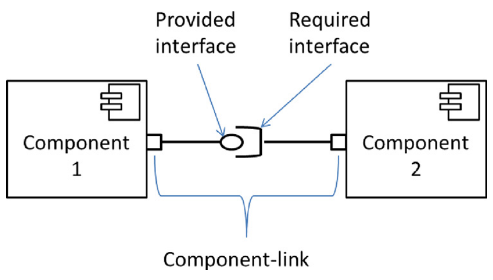


Fig. 8. An example of component-link.

A component may be linked with different components. A component may have links with a set of components in one product, and it may have other links with a different set of components in another product. Thus, the component-link variability is related to the component variability. This means that the identification of the component-link variability is based on the identified component variability. For instance, the existence of a link between *Component A* and *Component B* is related to the selection of *Component A* and *Component B* in the architecture. Thus considering a mandatory link is based on the occurrence of the linked components, but not on the occurrence in the architecture of products. In this way, a mandatory link is defined as a link that should be occur in the architecture configuration as well as the linked components are included in the configuration. To identify the component-link variability, we proceed similar to Algorithm 3 in terms of identifying the union and the intersection. For each architectural component, we collect the set of components that are connected to it in each product. The intersection of the sets extracted from all the products determines all mandatory links for the given component. The other links are optional ones.

5. Identifying architecture dependencies

In the previous sections, we identified component and configuration variability. In this section, we complete the identification of SPLA by recovering the architectural dependencies.

5.1. Identification of dependencies related to feature variability

The most common types of dependencies can be of five kinds: alternative, OR, AND, require, and exclude dependencies. To identify these dependencies, we rely on the same concept lattice generated in Section 4.2.1. In the lattice, each node groups a set of components representing the intent (e.g. *Com 5* and *Com 2*) and a set of architectural configurations representing the extent (e.g. *Arch 2*). The configurations are represented by paths starting from their concepts to the lattice concept root. The idea is that each object is generated starting from its node up going to the top. This is based on sub-concept to super-concept relationships (c.f. Section 2.1.3). This process generates a path for each object. A path contains an ordered list of nodes based on their hierarchical distribution; i.e. sub-concept to super-concept relationships).

The extraction of these paths is based on two steps. The first one is node numbering. This is done using Breadth First Search (BFS) algorithm (Cormen et al., 2009). In this context, BFS is used to identify a tree representation of a given graph. Starting from the root node (i.e. the top), BFS visits the nodes at distance *1*, then it visits the nodes at distance 2 and so on. Fig. 9 shows the process of how BFS orders the nodes, where node numbering refers to the distance of visiting a given node and ∞ denotes unvisited nodes. In the second step, starting from a node that holds an extent (e.g. *Arch 1*), we go up through links guiding us to nodes that carry a lower numbering and so on. This is recursively repeated,

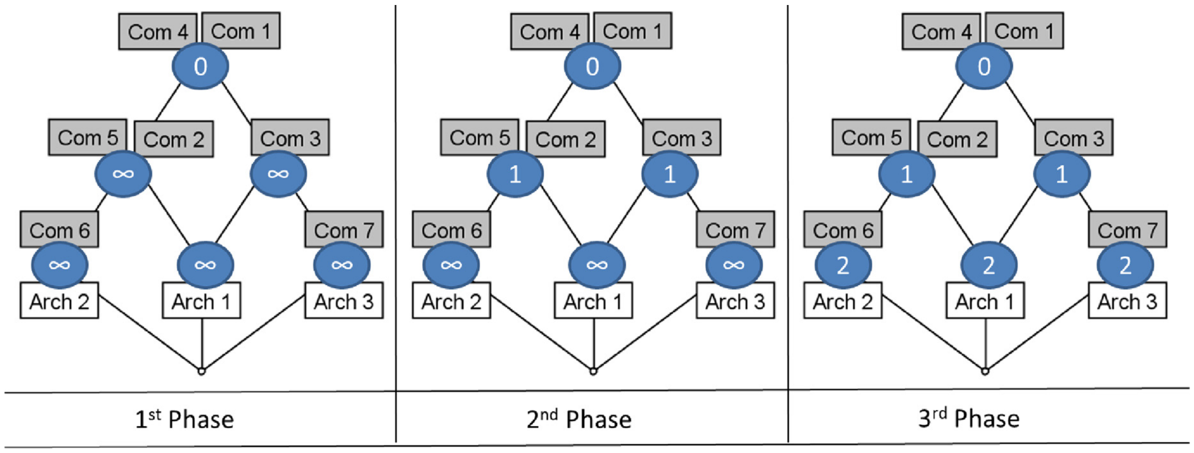


Fig. 9. An example of BFS process.

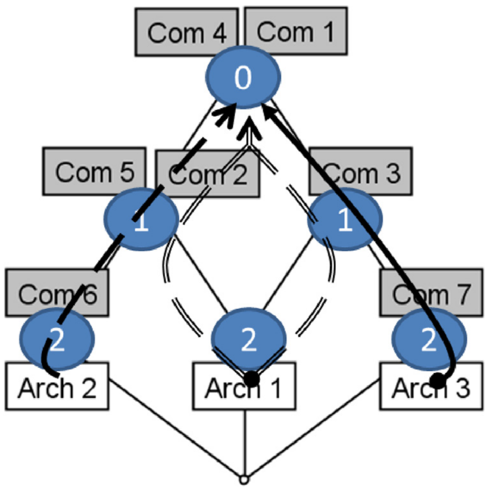


Fig. 10. An example of paths extracted from FCA lattice.

Algorithm 4: Identifying exclude pairs.

```
Input: All Pairs of Lattice Nodes and Paths($Pairs, Paths$)
Output: A Set of Pairs Having Exclude Dependency($ED$)
$ED = \emptyset$;
$Search for pairs having an exclude$
for each $pair \in Pairs$ do
    $isFound$ = false;
    for each $path \in Paths$ do
        if $path.contains(pair)$ then
            $isFound$ = true;
            break;
    end
    $If the pair is not found in the paths$
    if $isFound == false$ then
        $ED = ED \cup pair$ ;
end
return $ED$
```

where the termination condition is reaching the node having *0* numbering. Fig. 10 presents the paths identified in our example. There are three paths respectively presented by solid, dashed and double dashed arrows. For instance, the path corresponding to *Arch 1* includes the node of *Com 3*, the node of *Com 5* and *Com 2* and the node of *Com 1* and *Com 4*. According to these paths, we propose extracting the dependencies between each pair of nodes as follows.

5.1.1. Required dependency identification

Required dependency refers to the obligation selection of a component to select another one; i.e. *Component B* is required to select *Component A*. Based on the extracted paths, we analyze their nodes by identifying parent-to-child relation (i.e. top to bottom). Thus, node *A* requires node *B* if node *B* appears before node *A* in all paths, i.e., node *A* is a sub-concept of the super-concept corresponding to node *B*. In other words, to reach node *A* in any path, it is necessary to traverse node *B*. For example, if we consider the lattice in Fig. 7, *Com 6* requires *Com 2* and *Com 5* since *Com 2* and *Com 5* are traversed before *Com 6* in all paths including *Com 6* and linking root node to object nodes.

5.1.2. Exclude and alternative dependencies identification

Exclude dependency refers to the antagonistic relationship; i.e. *Component A* and *Component B* cannot occur in the same architecture. This dependency is identified based on the extracted paths. A node is excluded with respect to another node if they never appear together in any of the existing paths; i.e. there is no sub-concept to super-concept relationship between them. This means that there exists no object containing both nodes. For example, if we consider the lattice in Fig. 7, *Com 6* and *Com 7* are exclusives since they never appear together in any of the lattice paths. Algorithm 4 presents the procedure of extracting pairs of nodes that have the exclude dependency.

Alternative dependency generalizes the exclude one by exclusively selecting only one component from a set of components. It can be identified based on the exclude dependencies. Indeed, a set of nodes in the lattice having each an exclude dependency with all other nodes forms an alternative situation. For example, if node *A* is excluded compared to nodes *B* and *C* on the one hand, and node *B* is excluded compared to node *C*, on the other hand, then the group of *A, B* and *C* forms an alternative group.

5.1.3. AND dependency identification

AND dependency is the bidirectional form of the required one; i.e. *Component A* requires *Component B* and vice versa. More generally, the selection of one component among a set of components requires the selection of all the other components. According to the built lattice, this dependency is identified when a group of components is grouped in the same concept node in the lattice; i.e. all components grouped in this node should be selected together and not only a part of its components. For example if we consider the

Algorithm 5: Identifying OR-groups.

```
Input: All Pairs (ap), Require Dependencies (rd), Exclude
       Dependencies (ed) and Alternative Dependencies (ad)
Output: Sets of Nodes Having OR Dependencies (orGroups)
$Remove pairs and groups having required, exclude and
alternative$
OrDep = ap.exclusionPairs(rd, ed, ad);
$Remove pairs having transitive required$
OrDep = orDep.removeTransitiveRequire(rd);
$Identify nodes that share some components$
ORPairsSharingNode = orDep.getPairsSharingNode();
$Process the dependencies among the unshared components$
for each p ∈ ORPairsSharingNode do
    if otherNodes.getDependency() == require then
        orDep.removePair(childNode);
    else if otherNodes.getDependency()= exclude || alternative
    then
        orDep.removeAllPairs(p);
end
orGroups = orDep.getPairsSharingOrDep();
return orGroups
```

lattice in Fig. 7, *Com 2* and *Com 5* are concerned by an AND dependency.

5.1.4. OR dependency identification

When components are concerned by an OR dependency, this means that at least one of them should be selected; i.e. the configuration may contain any combination of the components. Thus, in the case of absence of other dependency, any pair of components is concerned by an OR dependency. Accordingly, pairs concerned with required, exclude, alternative, or AND dependencies are ignored as well as those concerned with transitive require dependencies; e.g. *Com 6* and *Com 7* are ignored since they are exclusives.

The process of identifying the OR groups is as follows. Firstly, we check the relationships between each pair of nodes. Pairs that have required, exclude, alternative, and AND dependencies are ignored. All pairs having transitive require dependencies are also ignored. The reason for this exclusion is that these dependencies break the OR one. Then, the remaining pairs of nodes are assigned an OR dependency. Next, we analyze these pairs by testing the relation of their nodes. Pairs sharing a node need to be resolved (e.g. in Fig. 7, a pair of (*Com 5 - Com 2, Com 7*) and a pair of (*Com 5 - Com 2, Com 3*), where *Com 5 - Com 2* is a shared node). The resolution is based on the relation between the other two nodes (e.g. *Com 3* and *Com 7*). If these nodes have a require dependency, then we select the highest node in the lattice (i.e. the parent causes the OR dependency to its children). If the dependency is excluded or alternative, then we remove all OR dependencies (i.e. an exclude dependency violates an OR one). In the case of sharing an OR dependency, the pairs are grouped to one OR dependency. AND dependency will not occur in this case according to AND definition. Algorithm 5 shows the procedure of identifying groups of OR dependency.

5.2. Identification of dependencies related to optional component distribution

These dependencies reflect association rules between optional components. Association rules refer to the frequency of co-occurrences between two groups of components. These can be used to discover interesting correlations between a large number of optional components. For instance, if an architectural configuration contains *Component A* and *Component B*, it has a probability of *70%* of also containing *Component C* and *Component D*. We rely on FP Growth algorithm (Han et al., 2006) to mine the association rules. Each architectural configuration is considered as a transaction and each component is an item that can exist in such a configuration. For example, in Fig. 7, we see that if a configuration contains *Com 1, Com 2*, and *Com 4*, there is a *100%* probably that it also contains *Com 5*. In addition to association rules, we recover the ratio of component occurrences in the products (e.g. *Component A* has occurred in *80%* of products). These can be used to provide information about which components are frequently used in the products. Fig. 7 shows, for example, that *Com 6* and *Com 3* are present respectively in *33%* and *67%* of the configurations.

6. Identification of groups of variability

In the previous steps, mandatory and optional components, as well as the dependencies among them are identified. However, understanding a large number of dependencies is a challenge facing software architects. Furthermore, some of these are overlapping dependencies. This means that many dependencies represent constraints on a shared set of components (e.g. a component has an OR dependency with components having AND dependencies among themselves). Such dependencies need to be hierarchically represented in a tree form, in order to facilitate the task of software architects (e.g. similar to feature model).

6.1. Variability between groups of dependencies

The idea is to identify dependencies among groups of dependencies. For example, a group of components holding an alternative dependency can have an OR dependency with another group holding an AND one. To identify these dependencies, our considerations are as follows. Since an AND can be considered as one coherent entity, it is not allowed for its components to partially have internal dependencies, e.g. dependent through an OR with other groups. This implies that all components belonging to an AND group should have the dependency. For alternative and OR dependencies, it is allowed to take AND ones as a member. In addition, the internal dependencies between alternative and OR dependencies are allowed. In other words, an alternative dependency can be a member an OR dependency and vice versa. According to that, the AND dependency has a high priority to be added before the others while OR and alternative have the same priority.

6.2. The identification algorithm

Algorithm 6 describes the procedure to identify the hierarchical tree. Firstly, we start from the root of the tree by directly connecting all mandatory components to it. At this stage, the tree does not have a hierarchy. Then, we add optional components based on their relationships. Groups of components having AND dependencies are added by creating an abstract node that carries out these components. The relation between the parent and the children is an AND dependency. Next, alternative dependencies are represented by an abstract node that carries out these components. Next, OR dependencies are applied by adding an abstract node as a parent to components having an OR. In the case where the relation is between a set of components having AND dependency as well as alternative dependency, the connection is made with their abstract nodes (i.e. the abstract nodes corresponding to the AND dependency and the alternative one become children of the OR parent). Next, the remaining components are directly added to the root with optional notation. Finally, the cross-tree dependencies are added (i.e. required and exclude ones).

Algorithm 6: Identifying hierarchical representation.

```
Input: Sets of Dependencies
       (OR, AND, Require, Exclude, Alternative) and Mandatory
       and Optional Components (MC, OC)
Output: A Tree (tree)
$Adding mandatory components to the root node$
tree.root.addChildren(MC);
$Adding AND groups to the root node$
for each and ∈ AND do
    tree.addChild(and);
end
$Adding OR groups to the tree$
for each or ∈ OR do
    for each node ∈ or do
        $Check if the OR contains a member composed of an
        AND group$
        if AND.isContiant(node) then
            $Remove the AND from the root and add it as a
            child in the OR $
            tree.remove(node);
            nodeOR.addChildren(node);
        else
            nodeOR.addChildren(node);
        end
        tree.addChild(nodeOR);
    end
end
$Adding alternative groups to the tree$
for each alt ∈ Alternative do
    for each node ∈ alt do
        $Check if it is a member in any already added group$
        if OR.isContiant(node) then
            break;
        else if AND.isContiant(node) then
            tree.remove(node);
            nodeAlt.addChildren(node);
        else
            nodeAlt.addChildren(node);
        end
        tree.addChild(nodeAlt);
    end
end
$Add the rest of the components as optional$
tree.addChildren(OC.getRemainingOptional());
$add cross tree dependencies$
tree.addExcludeCrossTree(Exclude);
tree.addRequireCrossTree(Require);
return tree
```

7. Evaluation results

7.1. Evaluation design

7.1.1. Data collection

We select two sets of product variants. These sets are Mobile Media[2] (MM) and Health Watcher[3] (HW). We select these products due to the availability of their source codes, the availability of architectural model for Mobile Media. This is way they were used in many published research papers such as Tizzei et al. (2012); Eyal Salman et al. (2015). Our study considers 8 variants of MM and 10 variants of HW. MM variants manipulate music, video and photo on mobile phones. They are developed starting from the core implementation of MM. Then, the other features are added incrementally for each variant. HW variants are web-based applications that aim at managing health records and customer complaints. The size of each variant of MM and HW, in terms of classes, is shown in Table 3.

[2] Available at: http://ptolemy.cs.iastate.edu/design-study/#mobilemedia.
[3] Available at: http://ptolemy.cs.iastate.edu/design-study/#healthwatcher.

7.1.2. Research questions and evaluation method

Our evaluation aims to show how the proposed approach is applied to identify the architectural variability and validating the obtained results. To this end, we applied it to the collected case studies. We utilize the ROMANTIC approach (Kebir et al., 2012b) to extract architectural components from each variant independently. Then, the components derived from all variants are clustered to identify component variants. Next, we identify the architecture configurations of the products that are used as a formal context to extract a concept lattice. Then, we extract the mandatory and optional components as well as the dependencies among optional components. Next, we present the results of identifying groups of variability. Finally, we show the identified variability.

In order to evaluate the resulting architecture variability, we study the following research questions:

- **RQ1: What is the accuracy of the recovered architectural variability?** This research question focuses on measuring the precision of the resulting architecture variability. This is done by comparing it with a pre-existing architecture variability model.
- **RQ2: Are the identified dependencies correct?** This research question aims to measure the correctness of the identified component dependencies.

7.2. Results

7.2.1. Component-based architecture extraction

Table 4 shows the results of component extraction from each product variant independently, in terms of the number of components, for each product variant of MM and HW. We check the services provided by each group of classes based on their source code documentation. The results show that classes related to the same service are grouped into the same component. The difference in the numbers of the identified components in each product variant has resulted from the fact that each product variant has a different set of user requirements. On average, a product variant contains 6.25 and 7.7 main services respectively for MM and HW.

7.2.2. Identifying component variants

Table 5 summarizes the results of component variants in terms of the number of components having variants (NOCV), the average number of variants of a component (ANVC), the maximum number of component variants (MXCV) and the minimum number of component variants (MNCS). The results show that there are many sets of components sharing most of their classes. Each set of components mostly provides the same service. Thus, they represent variants of the same architectural component. Table 6 presents an instance of 6 component variants identified from HW, where *X* means that the corresponding class is a member in the variant. By analyzing these variants, it is clear that these components represent the same architectural component. In addition to that, we noticed that there are some component variants having the same set of classes in multiple product variants. For internal component variability, we provide the lattice that presents the distribution of classes composing the component variants, see Fig. 11. From this lattice, we can note that there are *8* common classes between the variants. These represent the implementation of shared services.

Table 3
Size of MM variants and HW ones.

Variant no.	1	2	3	4	5	6	7	8	9	10	Avg.
No. of classes in MM	25	34	36	36	41	50	60	64	X	X	43.25
No. of classes in HW	115	120	132	134	136	140	144	148	160	167	136.9

Table 4
Component extraction results.

Variant no.	1	2	3	4	5	6	7	8	9	10	Avg.	Total
No. of components in MM variants	3	5	5	5	7	7	9	9	X	X	6.25	50
No. of components in HW variants	6	7	9	10	7	9	8	8	7	6	7.7	77

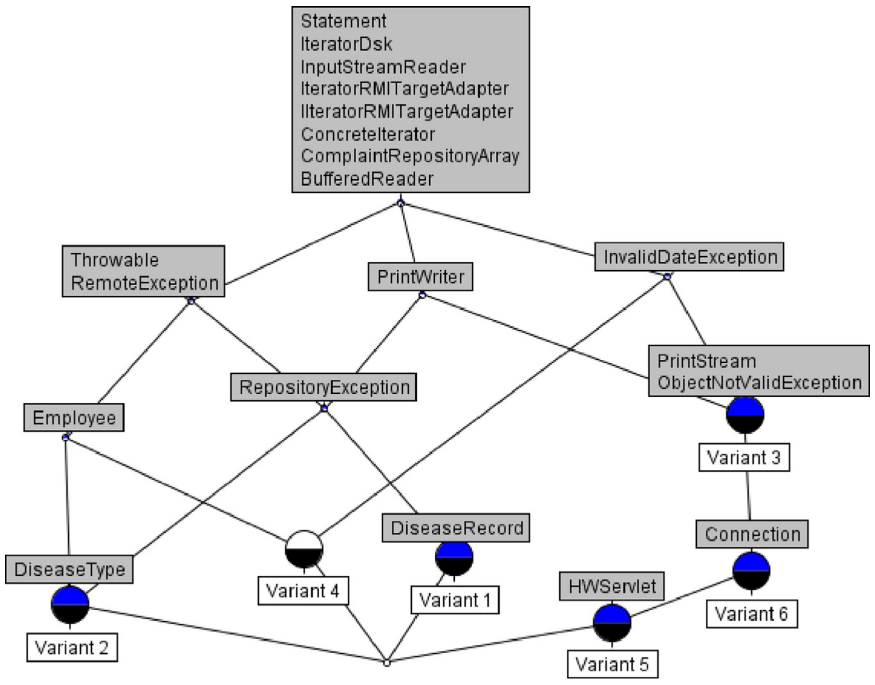


Fig. 11. The distribution of classes composing the component variants.

Table 5
Component variants identification.

Name	NOCV	ANVC	MXCV	MNCV
MM	14	3.57	8	1
HW	18	4.72	10	1

Additionally, the distribution of variable classes is easy to recognize. For example, the difference between *Variant3* and *Variant6* is that *Variant6* has an additional variable class (i.e. *Connection*).

7.2.3. Analyzing architecture configuration: communality and variability

The identification of component variants allows to identify the architecture configurations. Table 7 and Table 8 show respectively the configuration of MM and HW variants, where *X* means that the component is a part of the product variants. The results show that the products are similar in their architectural configurations and differ considering other ones. The reason behind the similarity and the difference is in the fact that these products are common in some of their user requirements and variable in some others. These architecture configurations are used as a formal context to

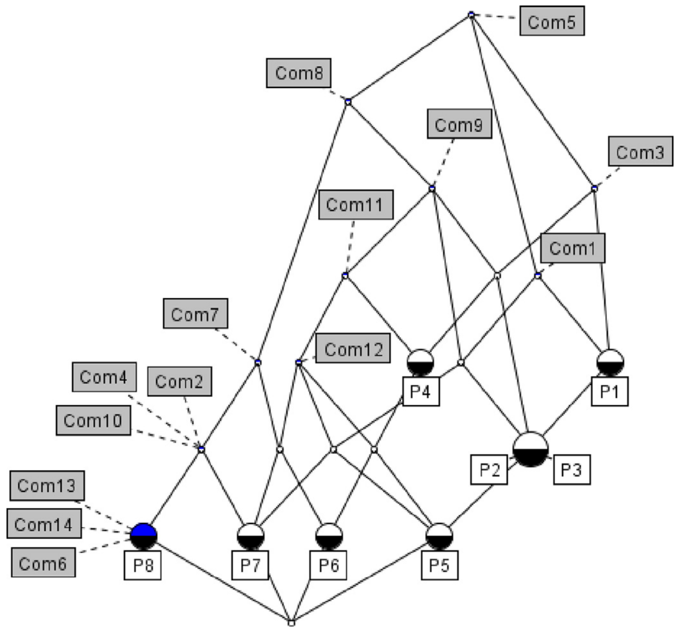


Fig. 12. The concept lattice of MM architecture configurations.

Table 6
Instance of 6 component variants.

Class Name	Variant 1	Variant 2	Variant 3	Variant 4	Variant 5	Variant 6
BufferedReader	X	X	X	X	X	X
ComplaintRepositoryArray	X	X	X	X	X	X
ConcreteIterator	X	X	X	X	X	X
DiseaseRecord	X					
IIteratorRMITargetAdapter	X	X	X	X	X	X
IteratorRMITargetAdapter	X	X	X	X	X	X
DiseaseType		X				
InputStreamReader	X	X	X	X	X	X
Employee		X		X		
InvalidDateException			X	X	X	X
IteratorDsk	X	X	X	X	X	X
PrintWriter	X	X	X		X	X
ObjectNotValidException			X		X	X
RemoteException	X	X		X		
PrintStream			X		X	X
RepositoryException	X	X				
Statement	X	X	X	X	X	X
Throwable	X	X		X		
HWServlet					X	
Connection					X	X

Table 7
Architecture configuration for all MM variants.

Variant No.	Com 1	Com 2	Com 3	Com 4	Com 5	Com 6	Com 7	Com 8	Com 9	Com 10	Com 11	Com 12	Com 13	Com 14
1	X		X		X									
2	X		X		X			X	X					
3	X		X		X			X	X					
4			X		X			X	X		X			
5	X		X		X			X	X		X	X		
6			X		X		X	X	X		X	X		
7		X		X	X		X	X	X	X	X	X		
8		X		X	X	X	X	X		X			X	X

Table 8
Architecture configuration for all HW variants.

Variant No.	Com 1	Com 2	Com 3	Com 4	Com 5	Com 6	Com 7	Com 8	Com 9	Com 10	Com 11	Com 12	Com 13	Com 14	Com 15	Com 16	Com 17	Com 18
1		X	X	X												X	X	X
2	X	X	X	X	X	X	X											
3	X	X	X	X	X	X	X	X	X									
4	X	X	X	X		X	X	X	X	X	X							
5		X	X	X		X	X				X	X						
6		X	X	X	X	X	X	X	X		X							
7		X	X	X	X	X			X	X	X							
8		X	X		X	X		X		X	X		X					
9		X	X		X	X		X			X			X				
10		X	X			X					X			X	X			

extract the concept lattice. We use the Concept Explorer[4] tool to generate the concept lattice. We give the concept lattices of MM and HW respectively in Fig. 12 and Fig. 13.

In Table 9, the numbers of mandatory and optional components are given for MM and HW, together with examples of their association rules and component occurrence ratios. An example of an association rule in MM is “when a configuration has *Com 5* and *Com 8*, it is *86%* likely to also contain *Com 9*”. The results show that there are some components that represent the mandatory architecture, while some others represent optional components.

7.2.4. Identifying components dependencies

The results of the identification of optional component dependencies are given in Table 10 and Table 11 respectively for MM and HW (*Com 5* from MM and *Com 2, Com 3* from HW are excluded since they are mandatory components). The dependencies are represented between all pairs of components in MM (where R= Require, E= Exclude, O= OR, RB = Required By, TR = Transitive Require, TRB = Transitive Require By, and A = AND). Table 12 shows a summary of MM and HW dependencies between all pairs of components. This includes the number of direct require dependencies (NRC), the number of exclude ones (NE), the number of AND groups (NOA), and the number of OR groups (NO). Alternative dependencies are represented as exclude ones. The results show that there are dependencies among components that help the architect to avoid creating invalid configurations. For instance, a design decision of AND components indicates that these components depend on each other, thus, they should be selected all together.

7.2.5. Identifying groups of variability

Based on the identified mandatory and optional components as well as the dependencies among the optional ones, we use the

[4] Presentation of the Concept Explorer tool is available in Yevtushenko (2000)

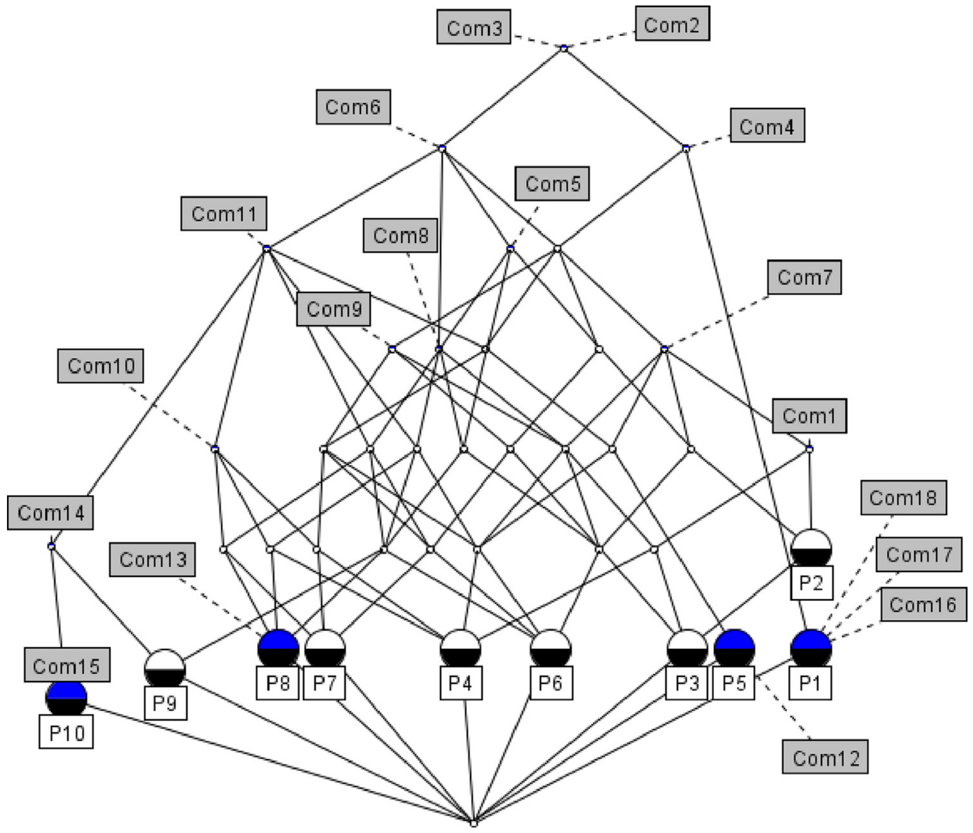


Fig. 13. The concept lattice of HW architecture configurations.

Table 9
Mandatory and optional components.

Product Name	MM		HW	
Mandatory	1		2	
Optional	13		16	
Some Association Rules	Com 5, Com 8 =>Com 9 : 86%		Com 2, Com 3, Com 11 =>Com 6 : 100%	
	Com 1, Com 5 =>Com 8, Com 9 : 80%		Com 2, Com 3, Com 4 =>Com 6 : 86%	
	Com 5, Com 12 =>Com 8, Com 9, Com 11 : 100%		Com 2, Com 3, Com 12 =>Com 4, Com 6, Com 7, Com 11 : 100%	
Some Component Occurrence Ratio	Component	Ratio	Component	Ratio
	Com 8	80%	Com 6	90%
	Com 12	38%	Com 7	50%
	Com 4	12%	Com 14	20%

Table 10
Component dependencies of MM.

	Com 1	Com 2	Com 3	Com 4	Com 6	Com 7	Com 8	Com 9	Com 10	Com 11	Com 12	Com 13	Com 14
Com 1	X		R		E	E				O	E	E	E
Com 2		X	E	A	RB	R	TR		A			RB	RB
Com 3	RB	E	X	E	E		O		E			E	E
Com 4		A	E	X	RB	R	TR		A			RB	RB
Com 6	E	R	E	R	X	TR	TR	E	R	E	E	A	A
Com 7	E	RB		RB	TRB	X	R	O	RB			TRB	TRB
Com 8		TRB	O	TRB	TRB	RB	X	RB	TRB	TRB	TRB	TRB	TRB
Com 9					E	O	R	X		RB	TRB	E	E
Com 10		A	E	A	RB	R	TR		X			RB	RB
Com 11	O				E		TR	R		X	RB	E	E
Com 12	E				E		TR	TR		R	X	E	E
Com 13	E	R	E	R	A	TR	TR	E	R	E	E	X	A
Com 14	E	R	E	R	A	TR	TR	E	R	E	E	A	X

Table 11
Component dependencies of HW.

	Com 1	Com 4	Com 5	Com 6	Com 7	Com 8	Com 9	Com 10	Com 11	Com 12	Com 13	Com 14	Com 15	Com 16	Com 17	Com 18
Com 1	X	TR		TR	R					ALT	ALT	E	ALT	ALT	ALT	ALT
Com 4	TRB	X		O	RB		RB			TRB	E	E	E	RB	RB	RB
Com 5			X	R	O	O	O		O	E	TRB		E	E	E	E
Com 6	TRB	O	RB	X	RB	RB	RB	TRB	RB	TRB	TRB	TRB	TRB	E	E	E
Com 7	RB	R	O	R	X	O	O		O	RB	E	E	E	E	E	E
Com 8			O	R	O	X	O		O	E	RB		E	E	E	E
Com 9		R	O	R	O	O	X		O	E	E	E	E	E	E	E
Com 10				TR				X	R	E	RB	E	E	E	E	E
Com 11			O	R				RB	X	RB	TRB	RB	TRB	E	E	E
Com 12	ALT	TR	E	TR	R	E	E	E	R	X	ALT	E	ALT	ALT	ALT	ALT
Com 13	ALT	E	TR	TR	E	R	E	R	TR	ALT	X	E	ALT	ALT	ALT	ALT
Com 14	E	E		TR	E		E	E	R	E	E	X	RB	E	E	E
Com 15	ALT	E	E	TR	E	E	E	E	TR	ALT	ALT	R	X	ALT	ALT	ALT
Com 16	ALT	R	E	E	E	E	E	E	E	ALT	ALT	E	ALT	X	A	A
Com 17	ALT	R	E	E	E	E	E	E	E	ALT	ALT	E	ALT	A	X	A
Com 18	ALT	R	E	E	E	E	E	E	E	ALT	ALT	E	ALT	A	A	X

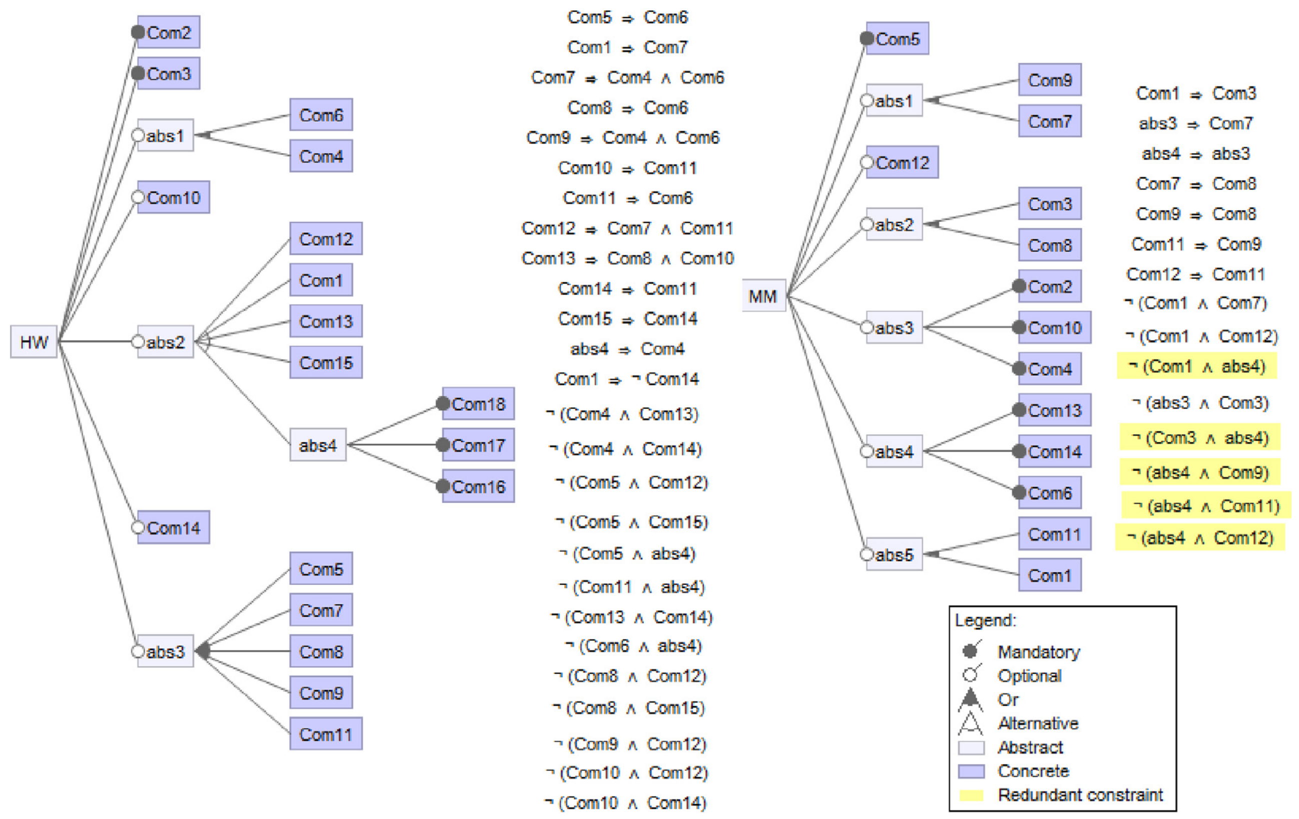


Fig. 14. The architecture variability trees of MM and HW.

Table 12
Summarization of MM and HW dependencies.

Name	NDR	NE	NA	NO
MM	17	20	6	3
HW	18	62	3	11

FeatureIDE[5] tool to visualize the tree in the form of a feature model. Fig. 14 shows the trees of both MM and HW, where Com_i => Com_j refers to a required dependency, and ¬ $(Com_i \wedge Com_j)$ refers to an exclude one.

[5] Presentation of the FeatureIDE tool is available in Thüm et al. (2014)

7.2.6. Architectural variability description

To the best of our knowledge, there is no architecture description language supporting all kinds of identified variability. The existing languages, like Hendrickson and van der Hoek (2007), are mainly focused on modeling component variants, links and interfaces, while they do not support dependencies among components such as AND-group, OR-group, and require. Thus, on the one hand, we use notations presented in Hendrickson and van der Hoek (2007) to represent the concept of component variants and links' variability. On the other hand, we propose notations inspired from feature modeling languages to model the dependencies among components. For understandability concern, we document the resulting components by assigning a name based on the most frequent tokens in their classes' names. This is automatically performed using the lexical analysis of classes' names. In many

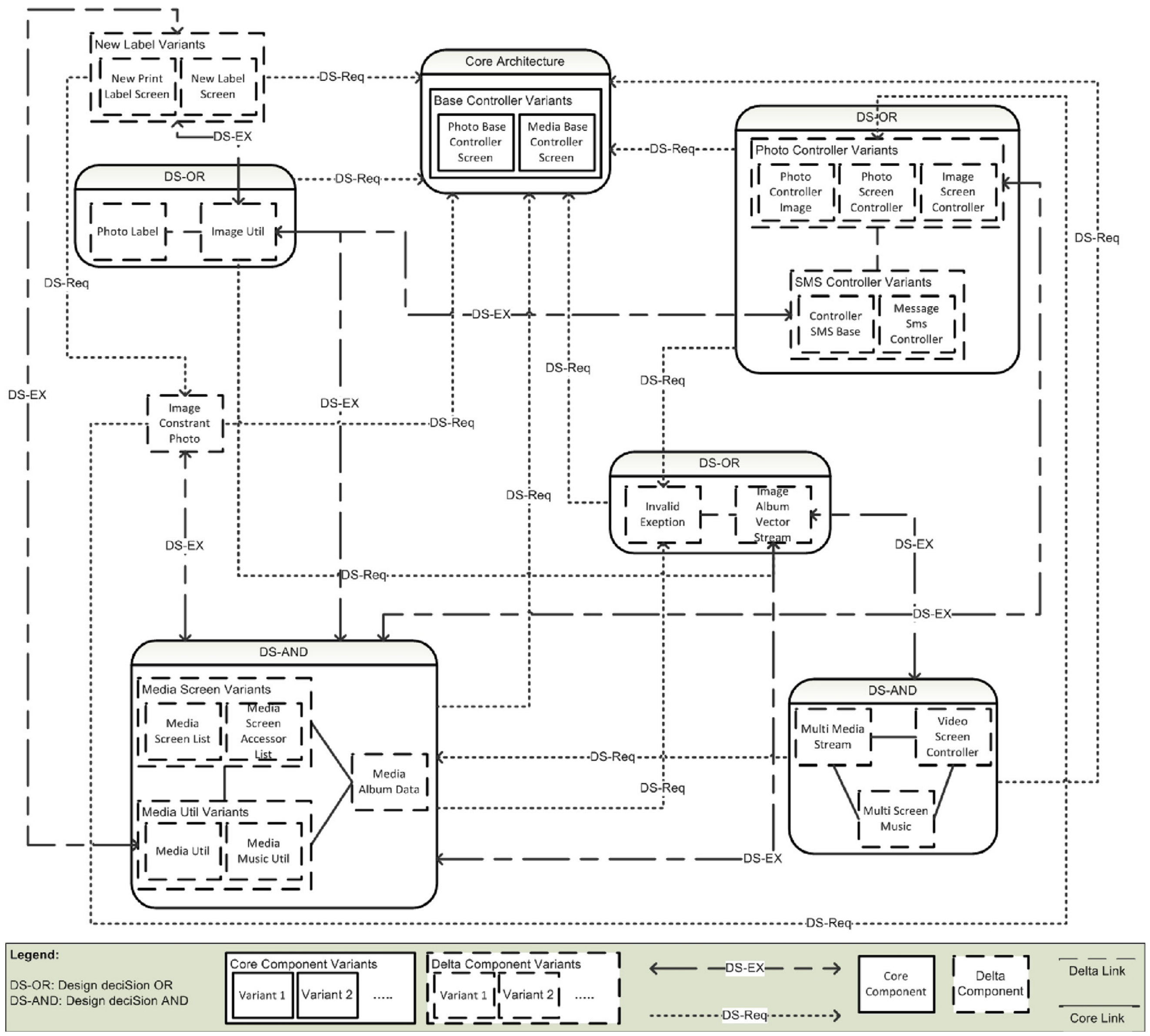


Fig. 15. Architectural variability model for MM.

cases, this was enough to produce meaningful names. In some cases, we had to manually adapt the selected tokens to produce a more appropriate component name. Fig. 15 shows the architectural variability model identified for MM variants, where the large boxes denote design decisions (dependencies). For instance, core architecture refers to components that should be selected to create any concrete product architecture. In MM, there is one mandatory component manipulating the base controller of the product. This component has two variants. A group of *Multi Media Stream*, *Video Screen Controller*, and *Multi Screen Music* components represents an AND design decision.

7.3. Answering research questions

7.3.1. RQ1: what is the accuracy of the recovered architectural variability?

In our case studies, MM is the only case study that has an available architectural-model containing some variability information. In Figueiredo et al. (2008), the authors presented the aspect oriented architecture for MM variants. This contains information about which products added components, as well as in which product a component implementation was changed (i.e. component variants). We compare both models to validate the resulting model. We firstly check the number of components constituting each architecture model. On the first hand, we find that our model is composed of 15 components, where each one has its variants. On the other hand, we find that the other model is composed of 14 components with their variants. Then, we map each component in our model to a corresponding component(s) in the pre-existing model based on services provided. If the mapped components both have variants, we consider that we correctly identify the variants of these components.

Fig. 16 shows the results in terms of precision and recall metrics for respectively (1) the identified components, (2) the identified component variants, and (3) the correct labeling of components as mandatory or optional. The architecture description of mobile media (reference architecture) given in Figueiredo et al. (2008) being aspect-based, each component of this architecture represents an implemented aspect. Two kinds of aspects are represented, functional/business and technical aspects. We note that components

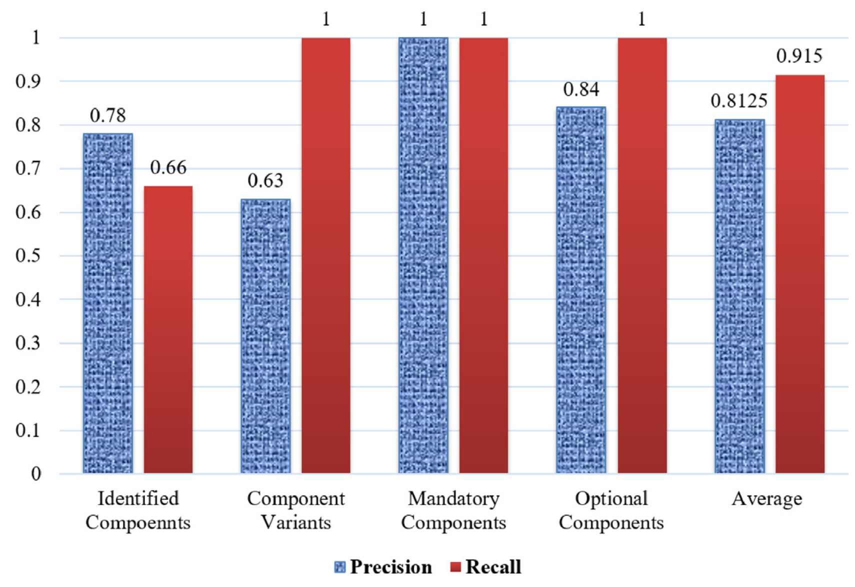


Fig. 16. The validation of MM architectural variability.

representing technical aspects (e.g., CommandListener component) are related to implementation details. Thus, at the architectural level, these components are typically embedded in components representing functional/business aspects. They should not appear at the same levels as the functional/business components. Software architecture recovery approaches conform to this definition of architecture and identify only the functional components (e.g., Allier et al., 2011; Weinreich et al., 2012a; von Detten et al., 2013). The ROMANRIC approach/tool that is used in our evaluation to recover architecture of single software identifies only functional/business components.

We compute the recall metric related to the identified components by considering all components described in the preexisting architecture model. In this context, the value of recall is 67%, i.e., we identified 67% of the reference components. These identified components represent 100% of the functional/business aspect components. This recall value is explained by the fact that the preexisting model describes additional components related to technical aspects that are not identified by ROMANTIC.

We compute recall metric related to the identified component-variants, and the correct labeling of components as mandatory or optional by considering only functional/business components, i.e., we ignore the technical components because they are not related to any architectural variability aspect.

Based on these results, the average precision and recall of the recovered architectural variability are respectively 81% and 91%.

7.3.2. RQ2: are the identified dependencies correct?

The identification of component dependencies is based on the occurrence of components. This method could provide additional dependencies compared to the real ones. For example, an AND dependency is generated for a group of components, when they are constantly existing in the product architectures. However, these components may coincidentally exist together, while they do not have an AND dependency. To evaluate the accuracy of this method, we manually validate the identified dependencies. This is based on the services provided by the components. For instance, we check if the component service(s) requires the service(s) of a required component and so on. As an example of a real required dependency is that *SMS Controller* component requires *Invalid Exception* one as it performs an input/output operations. On the other hand, *Image Util* component does not require *Image Album Vector Stream* one.

Fig. 17 shows the correctness of component dependencies identification in terms of precision and recall. On the one hand, based on the recall results (100% on average), our approach successfully identified all of architectural dependencies. On the second hand, based on the precision results (0.68% on average), our approach identified some additional superfluous dependencies that need to be ignored. The reason is that our approach is statistical-based. This means that the nature of the populations used as inputs affects the obtained results. Each product has a different impact on the results. This is based on the dependencies that its architecture could add to the recovered SPLA. To reduce the number of generated superfluous dependencies and improve the precision, the number of product variants to be studied should be increased.

7.4. Threats to validity

Two types of threats to validity concern the proposed approach. These are internal and external.

7.4.1. Threats to internal validity

The following three aspects have to be considered regarding the internal validity.

1. We use a static analysis technique to analyze the source code of the product variants. However, this analysis affects our results by two axes. The first one is that it does not address polymorphism and dynamic binding. Still in object-oriented, the most important dependencies are realized through method calls and access attributes. Thus, not dealing with polymorphism and dynamic binding does not have a high impact on the general results of our approach. The second effect is that the analysis does not differentiate the used and unused source code. This may introduce noise dependencies. However, this situation rarely exists in the case of well-designed and implemented software. In contrast, dynamic analysis addresses all of these limitations. Nevertheless, the challenge with dynamic analysis is to identify all use cases of software.
2. Our approach uses as input software architectures corresponding to multiple product variants of a software family. These architectures can be either the results of a forward-design or a recovery process. In the latter, many software architecture recovery (SAR) approaches can be used to identify the input of our SPLA recovery approach. Two criteria, however, need

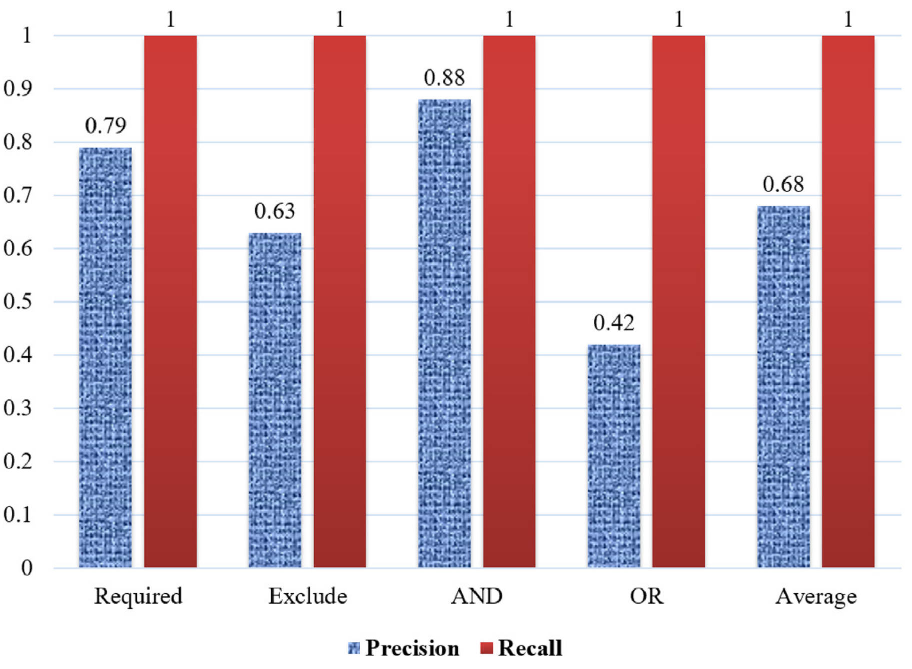


Fig. 17. The validation of component dependency correctness.

to be considered. The first one is related to the accuracy of the SAR approach. As the accuracy of the SPLA recovery process depends on the accuracy of the SAR approach, it is important to use a SAR approach with high accuracy. Some of these SAR approaches with high accuracy are cited in Garcia et al. (2013) and Lutellier et al. (2015). The second criterion is the architecture style. We consider as a valid input of our SPLA recovery approach architectures that are described in terms of component, where each component is implemented as a set of object-oriented classes. Considering these two criteria, manual adaptation may be needed to enhance the recovered architectures used as input of our SPLA recovery approach. For the evaluation of our approach, we relied on the ROMANTIC approach to individually recover software architectures of each single product variant based on the analysis of their source code. This may produce a threat to validity since that ROMANTIC does not provide 100% accuracy.

3. We evaluate ourselves the research questions. For the first question, the identified architecture variability is compared to the existing model. For the second question, we check the component services to evaluate the identified dependencies. The authors of this paper are familiar with the case studies since they studied it in many previous research projects, such as Eyal Salman et al. (2015) and Al-Msie'deen et al. (2014).

7.4.2. Threats to external validity

The two aspects that have to be considered regarding the external validity are:

1. The proposed approach is evaluated through product variants that are implemented in *Java*. However, the obtained results can be generalized for any object-oriented language. The reason behind this generalization is that all object-oriented languages (e.g. *C++* and *C#*) are structured in terms of classes and their relationships are realized through method calls, access attributes, etc.
2. We applied our approach on two case studies; Mobile Media and Health Watcher. We selected them due to the fact that they have a number of variants may reflect the need to migrate to product lines, the availability of their source codes and the availability of a representative model for architectural variability of Mobile Media. For these same reasons, these case studies were also used in many researches related to product lines.

8. Discussion

In this section, we discuss the pros and cons of the proposed approach.

8.1. Pros

We highlight the following advantages of the proposed approach:

1. It follows a fully-automatic process that does not need any manual interaction from software architects. Thus it can be applied for product variants where software architects are not available.
2. The only input required for our approach is source codes which is always available. Thus it can be applied for any product variants wherever the other software artifacts are available or not.
3. The approach is designed based on exploiting pre-proven algorithms, like breadth first search and formal concept analysis.
4. In the literature, no one recovers the whole SPLA. Our approach fills this gap by recovering the architecture variability concerning both component, configuration and component dependencies.
5. Our approach is scalable for large product variants knowing the complexity of the used algorithms. All of these algorithms require a polynomial time. For examples, breadth first search requires is $O(V+E)$, hierarchical clustering algorithm is $O(n^2)$, other algorithms with linear loops are $O(n)$. For generating FCA lattice, a scalable version of AOCpoest algorithm was proposed in Berry et al. (2014).

8.2. Cons

We identify the following limitations:

1. Component variants are identified based on the textual similarity between the classes composing the components. In some situations, a set of components may be implemented through

sets of classes that are textually similar, but they are completely unrelated. However, in the case of product variants developed using a copy-paste-modify technique, the modification is mainly composed of method overriding, adding or deleting, but the main services are always the same ones. In this context, we recommend to investigate other techniques, like Clone Detection and Latent Semantic Indexing.
2. Dependencies among components are identified based on component occurrences in the product architectures. This means that if the product variants included in the study are changed the resulting SPLA maybe also changed. For this point, we recommend to select product variants that cover a large number of dependencies. In addition, we propose to invest new source of information for the identification process, like feature model which contains the dependencies at the requirement level.
3. In our analysis, we only focus on component and configuration variability. This means that connectors are not taken into account during the reverse engineering process. We based ourselves on the fact that connectors are not considered as first class concepts in many architecture description languages such as Magee and Kramer (1996) Luckham (1996) Canal et al. (1999).
4. Refactored methods do not have a special manipulation. This may only affect our analysis process in interface variability identification step where we only focus on methods similarities. On the other hand, the other steps relied on class and component similarities. For example, during the identification of component variants, refactored methods do not affect our approach if they still belong to the same component. In addition, there is some refactoring cases that do not affect our approach. For instance, renamed methods or changed parameters do not have any impacts on the approach since the have the same implementations.

9. Related work

In this section, we discuss works related to reverse engineering software architectures research area. We provide our classification schema of the related approaches. Next, we provide a summarization of approaches aimed to identify SPLA.

9.1. Classification schema

In the literature, existing architecture identification approaches can be mainly classified based on three axes; the required input, the applied process and the obtained output.

9.1.1. The input of identification approaches

It can be source codes, documentations and human expert knowledge.

- Source code is used to identify the system structure, i.e., high cohesive and low coupling parts, by analysis relationships among classes. Classes are grouped into disjoint clusters, such that each cluster represents a component of an architecture view (Chardigny et al., 2008b; von Detten et al., 2013; Erdemir et al., 2011; Adjoyan et al., 2014). Some approaches allow the overlapping between the clusters (Mende et al., 2008; Shatnawi and Seriai, 2013; Shatnawi et al., 2016; 2015a).
- Software documentations are used to reduce the search space. For example, use cases are used to derive scenarios for dynamic analysis (Allier et al., 2011) or to extract a set of business functional components (Chardigny et al., 2008b). Some approaches used configuration files to extract information about the previous architecture (Weinreich et al., 2012b). Design documents are used to provide information about how components have been designed (Kolb et al., 2006). Class diagrams is used to identify the structural and behavioral dependencies among the system classes (Hamza, 2009).
- Human expert knowledge is used to select the main architecture view that needs to be the core of SPLA (Pinzger et al., 2004), to guide the identification by classifying the architecture units and identifying the dependencies among the components (Kang et al., 2005), or to add a new architectural information that is not immediately evident from source codes.
- A combination of different input resources can be used. For example, a combination is made to invest human expert knowledges to analyze and modify the resulted architecture (Erdemir et al., 2011). Another example is the usage of previous architectural information to guide of the identification approaches (Weinreich et al., 2012b; Pinzger et al., 2004).

9.1.2. The process of identification approaches

It is mainly composed of five aspects; algorithm applied, automation degree, process direction and analysis type.

- Several algorithms are used by existing approaches. These can be classified mainly into five types; search-based, clustering, FCA, clone detection, and authors defined heuristics. Existing approaches mainly used two kinds of search-based algorithms to partition classes. These are genetic algorithm (Kebir et al., 2012a) and simulating annealing (Chardigny et al., 2008b) or a combination of them Allier et al. (2011). Clustering algorithms used by von Detten et al. (2013); Boussaidi et al. (2012); Erdemir et al. (2011); Adjoyan et al. (2014). Formal Concept Analysis (FCA) is used as a clustering technique by Allier et al. (2009) to cluster object-oriented classes and by Hamza (2009) to partition software requirements into functional components. Clone detection algorithms are used to identify peaces of source codes that exist in many product variants to recover SPLA (Koschke et al., 2009; Frenzel et al., 2007). In Kolb et al. (2006), the authors used clone detection algorithms to identify similar components that could integrate the variability. In Mende et al. (2008); 2009), it is used to identify pairs of functions sharing most of their implementation. Some approaches proposed their own heuristic algorithms, instead of using predefined algorithms (Weinreich et al., 2012b; Kang et al., 2005; Pinzger et al., 2004).
- The automation of identification process refers to the degree in which this process needs human expert interactions. It can be manual, semi-automatic and full automatic process. Manual approaches fully depend on human experts. These ones only provide guides for the software architects (Wu et al., 2011; Langelier et al., 2005). Semi-automatic approaches need human expert recommendations to perform their tasks (Kang et al., 2005). In Pinzger et al. (2004), the authors relies on human experts to select the main architecture view that needs to be the core of SPLA and to manually analyze design documents. In Erdemir et al. (2011), software architects need to interact with the approach steps. Full-automated approaches do not have a high impact of human interactions on their results. For example, software architects need to determine some threshold values (Kebir et al., 2012a; Mende et al., 2008; Shatnawi et al., 2016; 2015a; Adjoyan et al., 2014).
- The process of identification approaches can be performed in three directions. These are top-down, bottom-up and hybrid directions. Top-down process partitions software requirements into architectural components (Hasheminejad and Jalili, 2015). Bottom-up process starts from source codes to extract software architectures (Boussaidi et al., 2012; Adjoyan et al., 2014; Kang et al., 2005; Koschke et al., 2009; Frenzel et al., 2007; Shatnawi et al., 2016; 2015a). Hybrid process refers to the analysis of

software requirements (top-down) and source codes (bottom-up) to identify the corresponding software architecture (Allier et al., 2009).

- Analysis type can be static, dynamic or lexical. Static analysis is performed without executing software (Pinzger et al., 2004; Kebir et al., 2012a; Weinreich et al., 2012b; Boussaidi et al., 2012; Shatnawi et al., 2016; 2015a). Dynamic analysis is performed by examining software at the run time (Allier et al., 2009). Lexical analysis refers to the textual analysis of source codes (Koschke et al., 2009; Frenzel et al., 2007; Mende et al., 2008; 2009; Kolb et al., 2006; Shatnawi et al., 2016; 2015a).

9.1.3. *The output of identification approaches*

It can be software architecture for single software or multiple software (SPLA).

- Most of the existing approaches identify the software architecture of single software (Erdemir et al., 2011; Hamza, 2009; Kebir et al., 2012a; Allier et al., 2011; Weinreich et al., 2012b). Some approaches provide a documentation about the extracted architecture (Ducasse and Pollet, 2009; von Detten et al., 2013). Some others support hierarchical architecture by providing a multi-layer architecture (Boussaidi et al., 2012). Some approaches tackled the problem of refactoring object-oriented code to be conform to a recovered component-based architecture. As a result, the recovered architecture can be implemented in component-based language such as OSGI, SOFA, etc. Alshara et al. (2015) and Alshara et al. (2016) are examples of these approaches.
- SPLA is recovered in terms of identifying component variants (Kolb et al., 2006; Koschke et al., 2009; Frenzel et al., 2007), component variability (Pinzger et al., 2004; Kang et al., 2005; Koschke et al., 2009; Frenzel et al., 2007), and dependencies between components (Kang et al., 2005). Some approaches identified reusable components from the source code of a set of product variants (Mende et al., 2008; 2009).

Table 13 presents the classification results of some related approaches based on the proposed classification schema. In Lutellier et al. (2015) and Garcia et al. (2013), the authors evaluated and compared some of the existing architecture recovery approaches. This aims to allows software architects to select the appropriate recovery approach.

9.2. *SPLA identification approaches*

Existing approaches suffer from two main limitations. The first one is that the architecture variability is partially addressed since they recover only some variability aspects, no one recovers the whole SPLA. The second one is that they are not fully-automatic since they rely on expert domain knowledge which is not always available.

In Koschke et al. (2009), an approach aiming to recover SPLA was presented. It identifies component variants based on the detection of cloned code among the products. However, the limitation of this approach is that it is semi-automated, while our approach is fully automated. Moreover, it does not identify dependencies among the components. In Kang et al. (2005), the authors presented an approach to reconstruct Home Service Robots (HSR) products into an SPL. Although this approach identifies some architectural variability, it has some limitations compared to our approach. For instance, it is specialized on the domain of HSR as the authors classified, at an earlier stage, the architectural units based on three categories related to HSR. These categories guide the identification process. In addition, they only rely on the feature modeling language (hierarchical trees) to realize the identified variability, this is not efficient since it is not able to

Table 13
The results of related work classification.

Approach	Input			Process				Output	
	Code	Doc.	Exp. knowledge	Algorithm	Automation	Direction	Analysis	Single software	SPLA
Our approach	X			FCA, clustering & BFS	Full	Bottom-up	Static		Complete SPLA
Chardigny et al. (2008b)	X	X		Simulating annealing	Full	Bottom-up	Static	X	
von Detten et al. (2013)	X			Clustering	Full	Bottom-up	Static	X	
Erdemir et al. (2011)	X		X	Clustering	Semi	Bottom-up	Static	X	
Adjoyan et al. (2014)	X			Clustering	Full	Bottom-up	Statci	X	
Allier et al. (2011)	X	X		Genetic & simulating anneal.	Full	Hybrid	Static, Dyn.	X	
Mende et al. (2008)	X			Clone detection	Full	Bottom-up	Static, lex.		Reusable components
Shatnawi et al. (2016)	X			Frequent usage patterns	Full	Bottom-up	Static, lex.	X	
Weinreich et al. (2012b)	X	X		Authors heuristic	Full	Hybrid	Static		
Kolb et al. (2006)	X	X		Clone detection	Full	Bottom-up	Static, lex.		Com. variants
Hamza (2009)		X		FCA	Full	Top-down	Dyn.	X	
Pinzger et al. (2004)	X		X	Authors heuristic	Semi	Bottom-up	Static, dyn., lex.		Com. variability
Kang et al. (2005)	X		X	Authors heuristic	Semi	Bottom-up	Static		Com. variability & dep.
Kebir et al. (2012a)	X			Genetic & clustering	Full	Bottom-up	Static	X	
Allier et al. (2009)	X	X		FCA	Semi	Hybrid	Dyn.	X	
Koschke et al. (2009)	X			Clone detection	Semi	Bottom-up	Static, lex.		Com. variants & variability

represent the configuration of architectures. The software architect plays the main role to identify the architecture of each single product and the dependencies among components. In some cases, the software architect is not always available. The authors in Acher et al. (2011) proposed an approach to reverse engineering architectural feature model. This is based on the software architect's knowledge, the architecture dependencies, and the feature model that is extracted based on a reverse engineering approach presented in She et al. (2011). The idea, in Acher et al. (2011), is to take the software architect's variability point of view in the extracted feature model (i.e. still at the requirement level); this is why it is named architecture feature model. However, the major limitations of this approach are, firstly, that the software architect is not available in most cases of legacy software, and secondly that the architecture dependencies are generally missing as well. In Pinzger et al. (2004), an approach is presented to recover SPLA of a family of software product variants. For each software product, it recovers a set of architecture views from the source code. Then, based on the software architect and documentations, they select one representative architecture view. Next, architecture views corresponding to all software products are analyzed to identify the community and the variability. The authors select one architecture view that represents the core of SPLA. Then, variation points resulted from the other products are incrementally added.

Another approaches are presented to identify reusable components based on the analysis of object-oriented product variants instead of SPLA recovery. In Shatnawi and Seriai (2013), the authors presented an approach to extract reusable software components based on identifying the similarity between components identified independently from each software. This approach can be related only to the first step of our approach (i.e. grouping similar components). In Mende et al. (2008); 2009), the authors presented an approach that aims to identify reusable components based on a clone detection algorithm to identify pairs of functions that share most of their implementation. Levenshtein metric is then used to measure the exact lexical similarity between the functions. Next, the functions are considered as nodes in a graph, while links connecting them are the lexical similarity value. Finally, nodes, i.e. functions, are aggregated into a high level to identify a set of functions that can be formed as a component. In Duszynski et al. (2011), an approach was presented to visually analyze the distribution of variability and commonality among the source code of product variants. The analysis includes multi-level of abstractions (e.g. line of code, method, class, etc.). This aims to facilitate the interpretation of variability distribution, to support identifying reusable entities.

10. Conclusion and future work

10.1. Conclusion

In SPLA, the variability is mainly represented in terms of components and configurations. In the case of migrating product variants to an SPL, identifying the architecture variability among the product variants is necessary to facilitate the software architect's tasks. Thus, in this paper, we proposed an approach to recover the architecture variability of a set of product variants. The recovered variability includes mandatory and optional components, the dependencies among components, the variability of component-links, and component variants. We relied on FCA to analyze the variability. Then, we proposed two heuristics. The former is to identify the architecture variability. The latter is to identify the architecture dependencies.

The proposed approach was validated through two sets of product variants derived from Mobile Media and Health Watcher software. The results showed that our approach can identify the architectural variability and the dependencies as well.

10.2. Future work

There are many future directions that can be indicated for this research. These include:

1. **Mapping the requirement variability and the architecture variability.** In the case of reengineering an SPL from product variants, mapping the identified architectural variability with the requirement variability is an important task. Thus, we plan to extend the identified SPLA by mapping it to the feature model which realizes the variability at the requirement level.
2. **Developing a visual environment.** The presented approach can be extended by providing a visual environment, such that software architects are allowed to interact with the approach at each step of the identification process and modify the obtained results when needed.
3. **Experimenting with a large number of case studies.** Our approach is statistical-based. This means that the quality and the size of the input populations (products variants) impact the resulting SPLA. For example, each product has its impact on the results by considering dependencies included in its architecture. These dependencies can be redundant between many product architectures. However, certain dependencies can be only included on some product architectures. Thus, we plan to extend the evaluation of the proposed approach by conducting more case studies which have different product variants.
4. **Validating our approach by third-party human experts.** We plan to evaluate our approach by third-party human experts.

References

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P., 2011. Reverse engineering architectural feature models. In: Software architecture. Springer, pp. 220–235.
Adjoyan, S., Seriai, A., Shatnawi, A., 2014. Service identification based on quality metrics - object-oriented legacy system migration towards SOA. In: The 26th international conference on software engineering and knowledge engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013., pp. 1–6.
Al-Msie'deen, R., Huchard, M., Seriai, A.-D., Urtado, C., Vauttier, S., 2014. Automatic documentation of [mined] feature implementations from source code elements and use-case diagrams with the revpline approach. International Journal of Software Engineering and Knowledge Engineering 24 (10), 1413–1438.
Allier, S., Sadou, S., Sahraoui, H., Fleurquin, R., 2011. From object-oriented applications to component-oriented applications via component-oriented architecture. In: Proceedings of 9th WICSA. IEEE, pp. 214–223.
Allier, S., Sahraoui, H.A., Sadou, S., 2009. Identifying components in object-oriented programs using dynamic analysis and clustering. In: Proceedings of the 2009 conference of the center for advanced studies on collaborative research. IBM Corp., pp. 136–148.
Alshara, Z., Seriai, A.-D., Tibermacine, C., Bouziane, H.L., Dony, C., Shatnawi, A., 2015. Migrating large object-oriented applications into component-based ones: Instantiation and inheritance transformation. In: Proceedings of the 2015 ACM SIGPLAN international conference on generative programming: Concepts and experiences. ACM, New York, NY, USA, pp. 55–64. doi:10.1145/2814204.2814223.
Alshara, Z., Seriai, A.-D., Tibermacine, C., Bouziane, H.L., Dony, C., Shatnawi, A., 2016. Materializing architecture recovered from object-oriented source code in component-based languages. In: Proceedings of European conference on software architecture. Copenhagen, Denmark.
Berger, C., Rendel, H., Rumpe, B., 2010. Measuring the ability to form a product line from existing products. In: Proceedings of the fourth international workshop on variability modelling of software-intensive systems, pp. 151–154.
Berry, A., Gutierrez, A., Huchard, M., Napoli, A., Sigayret, A., 2014. Hermes: a simple and efficient algorithm for building the aoc-poset of a binary relation. Annals of Mathematics and Artificial Intelligence 72 (1-2), 45–71.
Boussaidi, G., Belle, A., Vaucher, S., Mili, H., 2012. Reconstructing architectural views from legacy systems. In: 2012 19th working conference on reverse engineering (WCRE), pp. 345–354. doi:10.1109/WCRE.2012.44.
Canal, C., Pimentel, E., Troya, J.M., 1999. Specification and refinement of dynamic software architectures. In: Software architecture. Springer, pp. 107–125.
Chardigny, S., Seriai, A., Oussalah, M., Tamzalit, D., 2008a. Extraction of component-based architecture from object-oriented systems. In: Proceedings of 7th WICSA,. IEEE, pp. 285–288.
Chardigny, S., Seriai, A., Oussalah, M., Tamzalit, D., 2008b. Search-based extraction of component-based architecture from object-oriented systems. In: Software architecture. Springer, pp. 322–325.
Clements, P., Northrop, L., 2002. Software product lines: Practices and patterns.
Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2009. Introduction to algorithms. MIT Press.

DeBaud, J.-M., Flege, O., Knauber, P., 1998. Pulse-dssa-a method for the development of software reference architectures. In: Proceedings of the third international workshop on software architecture. ACM, pp. 25–28.

von Detten, M., Platenius, M.C., Becker, S., 2013. Reengineering component-based software systems with archimetrix. Software & Systems Modeling 1–30.

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K., 2013. An exploratory study of cloning in industrial software product lines. In: Software maintenance and reengineering (CSMR), 2013 17th European conference on. IEEE, pp. 25–34.

Ducasse, S., Pollet, D., 2009. Software architecture reconstruction: A process-oriented taxonomy. Software Engineering, IEEE Transactions on 35 (4), 573–591.

Duszynski, S., Knodel, J., Becker, M., 2011. Analyzing the source code of multiple software variants for reuse potential. In: Proceedings of WCRE. IEEE, pp. 303–307.

Erdemir, U., Tekin, U., Buzluca, F., 2011. Object oriented software clustering based on community structure. In: 2011 18th Asia pacific software engineering conference (APSEC). IEEE, pp. 315–321.

Eyal Salman, H., Seriai, A.-D., Dony, C., 2015. Feature-level change impact analysis using formal concept analysis. International Journal of Software Engineering and Knowledge Engineering 25 (01), 69–92.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Dantas, F., et al., 2008. Evolving software product lines with aspects. In: Proceedings of 8th ICSE. IEEE, pp. 261–270.

Frenzel, P., Koschke, R., Breu, A.P., Angstmann, K., 2007. Extending the reflexion method for consolidating software variants into product lines. In: 14th Working Conference on Reverse Engineering (WCRE). IEEE, pp. 160–169.

Ganter, B., Wille, R., 1996. Formal concept analysis. Wissenschaftliche Zeitschrift-Technischen Universitat Dresden 45, 8–13.

Garcia, J., Ivkovic, I., Medvidovic, N., 2013. A comparative analysis of software architecture recovery techniques. In: Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on. IEEE, pp. 486–496.

Gomaa, H., 2005. Designing software product lines with uml. In: Software engineering workshop - tutorial notes, 2005. 29th annual IEEE/NASA, pp. 160–216. doi:10.1109/SEW.2005.5.

Hamza, H.S., 2009. A framework for identifying reusable software components using formal concept analysis. In: Sixth international conference on information technology: New generations (ITNG), 2009. IEEE, pp. 813–818.

Han, J., Kamber, M., Pei, J., 2006. Data mining: concepts and techniques. Morgan Kaufmann.

Hasheminejad, S., Jalili, S., 2015. Ccic: Clustering analysis classes to identify software components. Information and Software Technology 57, 329–351.

Hendrickson, S.A., van der Hoek, A., 2007. Modeling product line architectures through change sets and relationships. In: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, Washington, DC, USA, pp. 189–198. doi:10.1109/ICSE.2007.56.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-oriented domain analysis (FODA) feasibility study. Technical Report. DTIC Document.

Kang, K.C., Kim, M., Lee, J., Kim, B., 2005. Feature-oriented re-engineering of legacy systems into product line assets–a case study. In: Software product lines. Springer, pp. 45–56.

Kebir, S., Seriai, A.-D., Chaoui, A., Chardigny, S., 2012a. Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. In: Proceedings of the fifth international C* conference on computer science and software engineering. ACM, pp. 1–8.

Kebir, S., Seriai, A.-D., Chardigny, S., Chaoui, A., 2012b. Quality-centric approach for software component identification from object-oriented code. In: Proceedings of WICSA/ECSA,. IEEE, pp. 181–190.

Kolb, R., Muthig, D., Patzke, T., Yamauchi, K., 2006. Refactoring a legacy component for reuse in a software product line: a case study. Journal of Software Maintenance and Evolution: Research and Practice 18 (2), 109–132.

Koschke, R., Frenzel, P., Breu, A.P., Angstmann, K., 2009. Extending the reflexion method for consolidating software variants into product lines. Software Quality Journal 17 (4), 331–366.

Krueger, C., 2002. Easing the transition to software mass customization. In: Software product-family engineering. Springer, pp. 282–293.

Langelier, G., Sahraoui, H., Poulin, P., 2005. Visualization-based analysis of quality for large-scale software systems. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering. ACM, pp. 214–223.

Linden, F.J.v.d., Schmid, K., Rommes, E., 2007. Software product lines in action: The best industrial practice in product line engineering. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Luckham, D., 1996. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events.

Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., Kroeger, R., 2015. Comparing software architecture recovery techniques using accurate dependencies. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, 2. IEEE, pp. 69–78.

Magee, J., Kramer, J., 1996. Dynamic structure in software architectures. ACM SIGSOFT Software Engineering Notes 21 (6), 3–14.

Mende, T., Beckwermert, F., Koschke, R., Meier, G., 2008. Supporting the grow-and-prune model in software product lines evolution using clone detection. In: 12th European conference on software maintenance and reengineering (CSMR). IEEE, pp. 163–172.

Mende, T., Koschke, R., Beckwermert, F., 2009. An evaluation of code similarity identification for the grow-and-prune model. Journal of Software Maintenance and Evolution: Research and Practice 21 (2), 143–169.

Nakagawa, E.Y., Antonino, P.O., Becker, M., 2011. Reference architecture and product line architecture: a subtle but critical difference. In: Software architecture. Springer, pp. 207–211.

Pinzger, M., Gall, H., Girard, J.-F., Knodel, J., Riva, C., Pasman, W., Broerse, C., Wijnstra, J.G., 2004. Architecture recovery for product families. In: Software product-family engineering. Springer, pp. 332–351.

Pohl, K., Böckle, G., Van Der Linden, F., 2005. Software product line engineering, 10. Springer.

Shatnawi, A., Seriai, A., Sahraoui, H., Al-Shara, Z., 2015a. Mining software components from object-oriented apis. In: Software reuse for dynamic systems in the cloud and beyond - 14th international conference on software reuse, ICSR 2015. Springer, pp. 330–347.

Shatnawi, A., Seriai, A., Sahraoui, H.A., 2015b. Recovering architectural variability of a family of product variants. In: Software reuse for dynamic systems in the cloud and beyond - 14th international conference on software reuse, ICSR2015, pp. 17–33. doi:10.1007/978-3-319-14130-5_2. Miami, FL, USA.

Shatnawi, A., Seriai, A.-D., 2013. Mining reusable software components from object-oriented source code of a set of similar software. In: Proceedings of IEEE 14th international conference on information reuse and integration (IRI), pp. 193–200. doi:10.1109/IRI.2013.6642472.

Shatnawi, A., Seriai, A.-D., Sahraoui, H., Alshara, Z., 2016. Reverse engineering reusable software components from object-oriented apis. Journal of Systems and Software.

She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K., 2011. Reverse engineering feature models. In: Proceedings of 33rd ICSE. IEEE, pp. 461–470.

Tavares, A.L.C., Valente, M.T., 2008. A gentle introduction to osgi. SIGSOFT Software Engineering Notes 33 (5), 8:1–8:5. doi:10.1145/1402521.1402526.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. Featureide: An extensible framework for feature-oriented software development. Science of Computer Programming 79, 70–85.

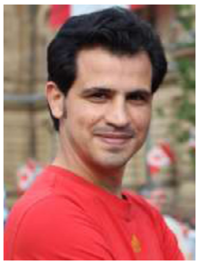
Tizzei, L.P., Rubira, C.M., Lee, J., 2012. An aspect-based feature model for architecting component product lines. In: Software engineering and advanced applications (SEAA), 2012 38th euromicro conference on. IEEE, pp. 85–92.

Weinreich, R., Miesbauer, C., Buchgeher, G., Kriechbaum, T., 2012a. Extracting and facilitating architecture in service-oriented software systems. In: Proceedings of WICSA/ECSA. IEEE, pp. 81–90.

Weinreich, R., Miesbauer, C., Buchgeher, G., Kriechbaum, T., 2012b. Extracting and facilitating architecture in service-oriented software systems. In: 2012 joint working IEEE/IFIP conference on software architecture (WICSA) and European conference on software architecture (ECSA), pp. 81–90. doi:10.1109/WICSA-ECSA.212.16.

Wu, Y., Yang, Y., Peng, X., Qiu, C., Zhao, W., 2011. Recovering object-oriented framework for software product line reengineering. In: Top productivity through software reuse. Springer, pp. 119–134.

Yevtushenko, A.S., 2000. System of data analysis "concept explorer". (In Russian) Proceedings of the 7th National Conference on Artificial Intelligence KII, Russia 79, 127–134.



Anas Shatnawi is a post-doctoral researcher at the laboratory for research on technology for ecommerce (LATECE) at University of Quebec at Montreal, Canada. He has a PhD degree from University of Montpellier, France. He obtained a M.Sc. in Computer Science from the Jordon University of Science and Technology in 2012, Jordan. His primarily interest is in software engineering with a particular focus on reengineering, reverse engineering, empirical software engineering, APIs reusability, software architectures, component and service development, and software product lines.

Abdelhak-Djamel Seriai is associate professor at University of Montpellier. He obtained his engineer degree in 1994. He obtained his PhD degree from University of Nantes, France in 2001. His research interests include software reuse, software architecture, software reengineering, reverse engineering, software component/service, software product line, software evolution, source code analysis, search-based algorithms, etc. He is author or co-author of more than 50 publications in international journals and conferences. He is the scientific editor of the first French book on "software evolution and maintenance". He is owner of the French scientific excellence reward from 2009 to 2013 and from 2014 to 2018 (award given by the French government in recognition of the scientific quality of a researcher).

Houari A. Sahraoui is professor at the department of computer science and operations research (GEODES, software engineering group) of University of Montreal. His research interests include software engineering automation, model-driven engineering, software visualization, and search-based software engineering. He has served as a program committee member in several IEEE and ACM conferences, as a member of the editorial boards of four journals, and as an organization member of many conferences and workshops.