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

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

a LIRMM, University of Montpellier, Montpellier, France
b DIRO, University of 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 SPL 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.

© 2016 Elsevier Inc. All rights reserved.

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 SPL 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). SPL 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.
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 architectural 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.

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 ROMANTIC1 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; composable, autonomy and specificity. These refer respectively to the

---

1 ROMANTIC: Re-engineering of Object-oriented systeM by ArchitecturE extracTion and migrated commponent based ones.

Please cite this article as: A. Shatnawi et al., Recovering software product line architecture of a family of object-oriented product variants, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.07.039
دریافت فوری متن کامل مقاله

امکان دانلود نسخه تمام متن مقالات انگلیسی
امکان دانلود نسخه ترجمه شده مقالات
پذیرش سفارش ترجمه تخصصی
امکان جستجو در آرشیو جامعی از صدها موضوع و هزاران مقاله
امکان دانلود رایگان ۲ صفحه اول هر مقاله
امکان پرداخت اینترنتی با کلیه کارت های عضو شتاب
دانلود فوری مقاله پس از پرداخت آنلاین
پشتیبانی کامل خرید با بهره مندی از سیستم هوشمند رهگیری سفارشات