On The Role of Architectures in Evolving Product-Line Software Systems

Haitham S. Hamza

Department of Computer Sc. & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115, USA
hhamza@cse.unl.edu

Abstract. Product-line engineering aims at developing a set of assets that provide the base to systematically evolving similar and related products in the domain. One of the core activities in any product-line engineering technique is the identification of the commonalities and variabilities among different products in the domain. These commonalities and variabilities are used to develop a base architecture from which the architecture of different products can be driven. We argue that, common practice for capturing and modeling commonalities and variabilities may lead to architectures that may not evolve effectively in response to changes in requirements or design decisions. Consequently, the role of architectures “decays” and their use in the development becomes marginal over time. In this paper, we first investigate some of the main issues that expedite the decay rate of product-line architectures. Then, we propose a vision that can enhance the role of product-line architectures not only for developing new products, but also for evolving existing products in response to requirements and design changes as well as domain changes.

Keywords: Product-line architectures, domain analysis, commonality analysis.

1 Introduction

In early 1960s, software community has realized the need for developing systematic software production techniques to overcome software complexity and to increase productivity [3]. However, unlike many other engineering disciplines, software development is inherently a people-oriented activity that is hard to be fully automated for mass production [5]. Research over the last four decades have stimulated several avenues to improve software productivity and to condense development time and reduce cost of software development. One such avenue is software reuse, where different software artifacts are developed to be used in developing similar and related systems. Several reuse communities have evolved, such as: Component-Based Software Engineering, software frameworks, patterns, and product-line architectures. These approaches vary in their reuse scope, scale and cost.
Product-Line Architectures (PLAs, for short) exploit reuse opportunities not only when developing a specific product, but also, when several similar or related products need to be developed. In such a case, identifying common features between these products and encapsulating them into reusable assets can provide a good opportunity for systematic reuse. The concept of developing a family of products by exploiting their commonalities have been under investigation since early 1970th [12] [13]. However, the increasing complexity of software systems coupled with the tied time-to-market constraints, have renewed much interest in this topic over the last few years.

Several engineering methods and techniques have been proposed to realize the concept of product-line architectures. Common to all these techniques is the core activity of identifying commonality and variabilities among different products in the domain; an activity that is known as commonality and variability analysis (C & V, for short). C & V analysis helps not only to identify potential reusable common assets among several products, but also to produce guidelines for developers to instantiate and evolve these common assets to develop new products. In a sense, (C&V) analysis classifies domain assets into stable (common) and unstable (variable) assets. We define a stable asset as one that can be adapted, extended, and reused without requiring major changes in its structures, and without inducing unnecessary cost. One might argue that systematic reuse is, indeed, realized truly by the stable assets of the domain.

C & V analysis, however, introduces several time-consuming as well as expensive activities compared to single-product development methods. This cost can be justified only if there is a feasible considerable reduction in cost and time in future developments. In practice, however, C & V analysis may not lead to assets that can used for a long time. In other words, the role of the developed assets decays over time. For one reason, this is because of the fact that customer requirements and technological advances change rapidly over time rendering many of the identified commonalities, and hence the architectures and other assets that employ these commonalities, to become obsolete in a short period of time.

In this paper, we take a closer look at the main aspects that can reduce the life-span of an architecture when evolving product-lines software systems. Then we highlight a vision that can enhance the role of architectures in developing product-lines systems.

The reminder of the paper is organized as following. Section 2 reviews existing techniques to develop commonality architectures. The concept of architecture life-time is discussed in Section 3. Section 4 presents the proposed approach. The paper concludes in Section 5.

2 Commonality Architectures

Domain Engineering and Domain Analysis methods are widely used to identify common features among different products in a domain, and to manifest these commonalities into reusable assets. Domain Engineering is defined as the process of developing reusable assets necessary to develop a family of applications.
within a defined domain. Domain Analysis (DA, for short), first introduced by Neighbors [11], is a core activity within domain engineering that captures the communalities within a given domain, and develops reusable artifacts that embody these communalities. This approach has gained a momentum over the last two decade evident by the proliferation of several DA methods, e.g. [2] [8] [9] [14] [15] [17] [18] [19]. A summary of some DA methods can be found in [1] and [10].

To simplify the discussion, we will consider a simple example of developing a family of telecommunication routers. In telecommunication networks, a source and a destination are connected through some intermediate routers, that are configured accordingly in order to establish the required connection. A router consists of several modules including: the switch unit and the control unit. Switch unit has input ports and output ports for receiving and sending signals, respectively. The control unit consists of the control circuits and algorithms required to configure the switch unit.

Using the notion of C&V analysis, one can develop a commonality architecture for electronic routers similar to that given in Figure 1. Different types of routers can be developed by adding additional components to this commonality architecture and by adjusting the features of the different components of the architecture. For example, one can adjust the size and the type of the “Memory” for the same type of routers, or change the type of the router itself by adding/removing the appropriate components. For instance, we can develop an ATM (Asynchronous Transmission Mode) router (Figure 1), or replace the ATM switch with an IP (Internet Protocol) switch to develop an IP router.

![Portion of the commonality architecture of an electronic router.](image-url)
3 Architecture Life-Time

Informally, the life-time of an architecture in the context of product-lines engineering can be defined as:

"the time after which it is no longer effective, time- and/or cost-wise, or no longer feasible to use the architecture for developing new products or evolving existing products in response to new changes in requirements and/or design decisions".

Ideally, we would like to develop architectures with an infinite life-time. In order to develop architectures with longer life-time, we must first understand what are the issues that can adversely affect the life-time of an architecture. We argue that most (C&V) analysis techniques focus on the “current” commonalities and variabilities among products of a specific domain, without considering potential changes that are likely to occur over time. In particular, we identify two main issues that are neglected by most existing (C&V) analysis techniques:

1. Product Evolution. After a single product is developed, new requirements may appear, existing requirement may change, or design decisions may need to be modified. In any case, the product needs to evolve to accommodate these changes. This can be achieved at the architecture level by adjusting the product’s architecture by either modifying elements, removing elements, adding elements, or a combination of the above. The real challenge in this case is that these modifications might create a chain of changes throughout the architecture, leading to two major problems. First, the resultant chain of changes may affect some of the elements that were developed from the commonality architecture of the product-line. Such changes can undermine the role of the common architectures, as any future changes to the common architecture cannot be systematically projected to the architecture of the evolved product. Even though advances in change management techniques aim to alleviate these kind of problems, in complex systems, however, change management cannot only be expensive and time-consuming but also an error prone activity. The other problem, and perhaps more important, is that the resultant chain of changes may also require considerable changes in the design of the product’s architecture, raising the likelihood of “redesigning” the architecture from scratch, which defeats the purpose of systematic reuse in PLAs.

2. Domain Evolution. Domains can evolve too. This fact is neglected in most existing analysis methods. Most existing (C&V) analysis techniques tend to assume that identified communalities are “stable”, i.e. they are enduring in the domain. However, this assumption does not always hold. In fact, several features that appear to be common to the products of the domain can become “unstable” over time, as they either require considerable changes or even vanish. As a result, PLAs developed based on these “unstable” communalities become unstable as well. To illustrate this point, consider the OEO (Optical-Electronic-Optical) converters shown in Figure 1, for example. OEOs seem to be common elements in any electronic router. However,
with the continuing advances in optical technology, future routers are expected to operate in the optical domain, where no OEOs are needed. In this case, the domain has not changed, indeed we are still interested in developing routers; however, technological advances have caused the domain to evolve. In complex architectures, accommodating such evolution may be very costly as several changes may be required.

The above discussion highlights two of the most important aspects that need to be addressed in commonality architectures. The first aspect is scalability. Commonality architectures must allow for an additive changes rather than invasive changes. The second aspect is related to abstraction. We believe that a higher abstraction level above the commonalities and variabilities abstractions is required in order to accommodate domain evolution. These two aspects have led us to rethink the way that (C&V) analysis should be conducted. Consequently, we develop a new approach to attain the two aspects discussed above.

![Diagram](image)

**Fig. 2.** The commonality architecture of an electrical router applying the new vision.

### 4 The Proposed Approach

It seems unrealistic to envision an approach that can lead to a timeless architecture for a given product-line system. What is realistic however is to develop approaches that can improve the life-time of architectures. Based on the discussion in Section 3, it is clear that the life-time of an architecture is directly related to its capability in accommodating both product evolution as well as domain evolution.
We argue that in order to accommodate both product and domain evolutions, we must focus on the core concepts that are common to the different products in the domain rather than simply considering the features that “currently” seem to be common among several products. Core concepts seem to be more stable and enduring compared to features and other elements. For example, in the analysis of electronic routers, we may identify the “storing” concept as a common concept among any routers. This concept can be realized by any means depending on the technology and the type of the router. Moreover, even when routers evolve from the electronic to the optical domain, the “storing” concept still valid, although the realization of the concept is quite different in both domains.

Thus, the new commonality architecture contains “Storing” as an element. The realization of the storage concept depends on the type of the router and should not trigger a new analysis or major changes in the old architecture of the electronic routers. Figures 2 and 3 shows portions of the architectures for the electronic and the optical routers, respectively, using the new approach. Note that the two architectures have several elements in common. In particular, they share all the core concepts of the domain such as “switching”, “management”, “conversion”, and “storing”.

On the other hand, when domains evolve, some concepts may need to be removed; new concepts need to be added; or both. To do so in large and complex architectures, there is always the danger of major changes due to the ripple effect of some changes. In the worst case, the overall architecture may collapse.

\[1\] In optical routers; the notion of Buffers cannot be explicitly realized because of the current lack of optical technology. Physically, signals are stored in the optical domain by being delayed through multiple fiber-delay loops (FDLs).
forcing the developer to develop the architecture from scratch. To avoid this problem, one may think of the commonality architecture as a collection of units instead of a set of individual concepts. For example, in Figure 3, two units are highlighted (in the dashed boxes), one unit encapsulates concepts related to the “switching” functionality, whereas the other unit consists of the concepts related to the “conversion” process.

Now, consider a simple domain evolution example, where a new optical device that can simultaneously switch and convert optical signal has become available. An example of such device is the Wavelength Exchanger Optical Crossbar (WOC) proposed in [7]. In such a case, the overall conversion unit is not needed. As shown in Figure 4, one can remove the conversion unit and replace the “Optical Switch” with the “WOC” device. Notice that this change has not affected the other elements in the architecture.

To realize the above approach, two main questions need to be answered: (1) How to identify the core common concepts of a domain? , and (2) How to group different concepts into units that can be added and removed with minimal impact on the rest of the architecture? Currently, we are developing an approach that applies the concepts of Software Stability Model (SSM) [4] to identify the common concepts of the domain. In order to separate and encapsulate concepts into different units, the approach uses the mathematical theory of Formal Concept Analysis (FCA) [16] along with a set of quantitative metrics, similar to those proposed in [6]. In the following, we first provide a brief background on the main concepts used in our approach, then we highlight the main activities in the approach.
4.1 A Brief Background

The proposed approach is based on two notions: the Formal Concept Analysis (FCA) theory [16] and the Software Stability Model (SSM) [4]. FCA is a mathematical framework that can be used to represent and analyze data and their relationships [16]. A formal concept is defined as a pair \((O, F)\), such that \(\beta(O) = F\) and \(\alpha(F) = O\), where \(G\) is a set of objects; \(F\) is a set of features; \(O \subseteq G\); and \(F \subseteq M\). \(\beta(O)\) is an operator that is defined as the set of features shared by all the objects in \(O\). Similarly, \(\alpha(F)\) is defined as the set of objects that share all the features in \(F\).

SSM is a generic layered approach for modeling software that classifies the classes of the system into three layers: Enduring Business Themes (EBTs): contains the enduring and core knowledge of the underlying business; Business Objects (BOs): contains classes that map the EBTs of the system into more concrete objects; and Industrial Objects (IOs): IOs are classes that map the BOs into concrete objects. In a banking system, for example, one possible EBT is “ownership”; without an “Ownership”, there is no account. “Account” is a BO, while a “SavingAccount” is a concrete “Account”, and hence it is an IO.

4.2 Main Activities

In the following, we provide a brief overview of the main activities of the proposed approach:

1. **Analysis Phase**: In this phase, first the requirements are analyzed using existing analysis and requirements engineering techniques. This step produces a set of Functional Requirements and Non-Functional Requirements. Next, the problem is analyzed using the concepts of SSM (Section 3) in order to identify the EBTs, BOs, and IOs of the system. In addition, a use-case model is developed and use-case scenarios are identified and specified. Finally, a set of core concepts in the domain are identified.

2. **Formal Concept Analysis Phase**: In this phase, we first construct the formal context of the system with \(G\) being the set of all classes in the system and \(M\) being the set of all use-cases. In the formal context, an \(X\) is placed in the intersection between a class and a use-case, if the class is a participant of the use-case. Next, we generate the formal concepts of the system. It should be noted that not all the generated formal concepts are relevant concepts. For example, some formal concepts may violate the structure of the SSM, and hence, they must be eliminated. The elimination of irrelevant concepts is done by identifying a set of filtering rules and examine each formal concept against these rules. A rule is a predefined constrain that should be satisfied by each concept for it to be considered relevant.

3. **Concept Encapsulation Phase** In this phase, we first match each core concept in the system with a set of use-cases that realizes this core concept. The set of formal concepts that realizes a given core concept in the system

\(^2\) http://www.st.cs.uni-sb.de/~lindig/ (online)
is called a “cluster”. We then measure the relevance of each formal concept in the system with respect to a given cluster using quantitative metrics. Four metrics are used: (1) **Coverage Percentage**: measures the percentage by which a concept covers a given concern. This metric is computed for each concept with respect to each concern in the system; (2) **Coupling Index**: measures the average interaction between a given concept and all concerns; (3) **Stability Index**: measures the level of stability of a given concept. That is, how enduring the concept is during the system life-cycle; (4) **Quality Factor**: gives weights to each index above to compute the overall quality of a given concept with respect to each concern. Based on these metrics, we can identify the best set of formal concepts that can realize (i.e. covers all the use-cases) of a given core concept in the system. The collection of selected formal concepts will constitute the a unit that represent a core concept in the architecture.

The main challenge in the proposed approach is that, some core concepts cannot be separated into units as they are scattered across the system, and hence, they cross-cut other units in the architecture. To solve this problem, we have introduced the notion of “Autonomous” concepts and “Distributed” concepts\(^3\). The former are concepts that can be isolated and encapsulated into stand-alone units, whereas the later are concepts that cross-cut other units, and hence, they cannot form a stand-alone unit. Based on this classification, we devised two approaches to handle each concept type when evolving the architecture.

5 Conclusions and Current Work

In this paper, we discussed some of the main issues that may impact the role of architectures in evolving product-lines software systems. It appears that most existing approaches for developing architectures in product-line systems do not explicitly consider the evolution of products and the evolution of the domain. This can lead to architectures that have short life-time in the development process. To overcome this problem, we proposed an approach in which *domain concepts* are identified and encapsulated to develop the commonality architecture of the domain rather relaying on features that appear to be common but can potentially change over time. This work still in its preliminary stages and further investigation and validations are certainly needed. However, we believe that the proposed approach holds the promise to develop more effective PLAs.

References


\(^3\) Distributed concepts can be viewed as aspects. However, we prefer to use different name to avoid the confusion between the two concepts, as each is handled in different ways.


