Quality-Driven Conformance Checking in Product Line Architectures

Femi G. Olumofin and Vojislav B. Mišić
University of Manitoba, Winnipeg, Manitoba, Canada

Abstract

Software product line are often developed through reengineering existing products and legacy applications. In such cases it is not uncommon for the behavioural and quality characteristics of individual product architectures to be inconsistent with those of the common architecture. Successful development of product lines dictate that those inconsistencies be resolved. The resolution process involves bringing the product architecture into structural, semantic and quality attribute-related congruence with the common architecture. Additional steps must be taken to ensure their continued conformance in order to facilitate subsequent maintenance and evolution activities. In this paper, we describe a simple design-time technique that aims to ensure that quality attribute responses of individual product architectures are in conformance with those of the common architecture. The technique is based on the concept of variation points.

1. Introduction

For more than a decade, software architecture has been steadily gaining importance as the most effective vehicle for the development of complex software-intensive systems. Architecture-based design offers unmatched flexibility and allows crucial insights to be obtained very early in the design cycle. Architectural abstraction avoids complex code level details while making component structures and interrelationships explicit. In this manner, the use of architecture facilitates human understanding of the system as well as reasoning about quality characteristics and attributes. It should come as no surprise, then, that the reengineering of existing systems and legacy applications—recovering their structure in order to develop new, functionally equivalent but improved systems—often focuses on recovering or reconstructing the architecture in the form of a product. Most such efforts are motivated by changes in quality attributes, such as extendibility and maintainability, rather than by the need for functional changes and enhancements [3, 10]. For example, consider a system that has undergone several maintenance cycles which included functionality enhancements. While the system itself may be in working order, the documentation complexity and, possibly, inconsistency make further maintenance difficult. The first thought would be to leave the system as it is and reconstruct the documentation only; but a better way is to disregard the documentation and recover the system architecture from the system itself. Oftentimes, architecture recovery is the first step towards reengineering the entire system.

All of the aforementioned advantages are even more important in the case of software product families or product lines [5]: sets of related yet distinct software-intensive systems developed from the same base architecture. In the product line approach, requirements or features common to all the products are used as the basis for the so-called *core architecture*, or CA. Requirements which are specific to some of the products only, but not all of them, are represented as *variation points* in the CA. (It is common to refer to the two sets of requirements as commonality, or commonalities, and variability, respectively.) Individual products are then developed to address the specific sets of requirements. In one approach, individual products are directly developed from the CA by replacing the variation points with product-specific component instances, called *variants*. This approach is often used in simpler cases – i.e., when the number of individual products and/or variation points is not high.

In an alternative approach, the CA is used to instantiate a number of separate *product architectures* or PAs, which correspond to individual products. The PA is created from the CA by exercising the built-in variation points. The actual products are, then, developed from the corresponding PAs. This *dual form* of representation of the architecture (i.e., CA and PA) is typical of the software product lines [5, 6].

Yet more problems arise when the product line development path involves the reuse of existing products. In most cases, existing products and legacy systems were built with little care (or none at all) for consistency and quality, thus encumbering the identification of commonalities and variability required for the product line approach. Once identified and specified, the CA and the individual PAs may differ significantly, in particular with regard to consistency and prioritization of quality attributes. Any inconsistencies and differences in the architectures recovered from the existing system must be resolved in the product line architectures – successful development of the reengineered system is contingent upon the design of both CA and PAs being quality attribute-driven and conflict-free.

In this paper, we present a design-time technique for maintaining conformance between the reengineered and evolving CA and individual product architectures. The technique is based on the concept of variation points, which are exploited in a systematic fashion in order to constrain the individual PAs to be consistent with the CA. While the approach described is particularly suited to reengineering product lines, its generality makes it also applicable for validation of product line architectures developed 'from scratch' as well as those developed using the revolutionary approach [2].

The paper is organized as follows. In Section 2, we briefly describe the challenges of ensuring quality conformance between the CA and the PAs, and discuss some earlier work that touches this issue. Section 3 introduce our technique based on variation points, together with a small example that illustrates the use of the technique. Finally, Section 4 summarizes the paper and highlights some open issues for further work.

2. Challenges and Related Work

As noted above, the product line architecture consists of a core architecture (CA) which is used as the basis for developing a number of individual product architectures (PAs). The CA is necessarily underspecified, while the individual PAs must be fully specified since the actual products will be derived from them. However, the set of quality attributes for a given PA may significantly differ from that of the underlying CA, and even priorities of different attributes may differ. To consider the interplay between the quality attributes of individual PAs and those of the CA, we need to start by considering the CA. The quality attribute goals in the CA are addressed through the so-called sensitivity and tradeoff points [1, 4]. A sensitivity point is an area of the architecture which determines the responses of at least one quality attributes. A tradeoff point is an area of the architecture which determines the responses of two or more quality attributes, usually in opposing ways. (Note that each tradeoff point is a sensitivity point by default.)

The problem lies, of course, in that the individual PAs have quality attributes and priorities of their own. Satisfying those attributes may cause conflict with the decisions made in the CA, thus compromising the quality attributes that should be common to both the CA and all PAs. Namely, the changes that fully specify an underspecified CA, and thus instantiate the particular PA, are made in an area with a variation point – the requirement specific to the PA but not present in the CA itself. If the variation point overlaps with a sensitivity point of the original CA, the corresponding quality attribute may be affected. If the variation point overlaps with a tradeoff point, several of the original attributes will be affected. Now, each of the individual PAs instantiates a particular variation point from the underlying CA in its own fashion. As a result, conformance checking between the CA and individual PAs is a complex process, and the problem is not made any easier by the fact that there may be quite a few PAs derived from a single CA.

Several authors have identified this problem in the context of architecture reengineering. In most cases, such reengineering is based on updating the 'as-designed' architecture of a system from the 'as-built' architecture reconstructed by reverse engineering. Once the architectural description of the existing system is accurately specified, it can be modified in order to fulfill the emergent quality goals of the new target system.

Bengtsson and Bosch present an iterative, scenario-based re-engineering method for transforming software architectures to provide desired quality attributes responses [3].

QADSAR [13] is a quality attributes scenario driven reverse engineering method for architectures of existing system; whose tool support is the ARMIN. The goal of a QADSAR reconstruction is to provide architectural description and information on architectural drivers to enable qualitative architectural analysis.

Stoermer *et al.* [12] provides a codification of six practice patterns for architectural reverse engineering. These patterns are described with a name, context of application, concise statement of problem in the context, an example illustration in an industrial context, and the expected solution/delivery from applying the pattern. The paper also describes some common approaches to reverse engineering, including tool supported approaches. The suitability of different approaches (and the accompanying tools) for use in the practice patterns is also discussed. The result of the analysis revealed the lack of adequate coverage for the practice pattern by the

existing approaches.

Finally, Tahvildari *et al.* [14] proposed a quality-driven software reengineering framework similar to that of Bengtsson and Bosch [3]. This framework is based on the use of desirable target-system qualities to define and guide the reengineering. According to the Stoermer's practice pattern catalogue [12], this framework may be categorized into the quality attribute changes practice pattern. In this pattern, legacy systems are re-engineered to improve some desired quality attributes responses, such as performance or maintainability.

3. Variation Point Concepts Usage

In order to ensure quality congruence between the common architecture and individual product architectures, both the existing and the emerging ones, we make use of the concept of variation points. Variation points are architectural placeholders for augmenting the CA with behavioural extensions. They are instantiated as concrete variants in individual product architectures. The sensitivity points are those architectural decisions that affect one or more quality goals [8]. For example, the encryption of sensitive message exchange between two components may improve the security quality of a software-intensive system. The architectural decision to introduce cryptographic components between the two communicating components is a sensitivity point intended to implement security insofar as message exchange between the two components is concerned.

Architectural decisions made in the process of defining the CA, and subsequently found to be sensitivity points to one or more quality attributes, continue to remain valid for individual product architectures. A possible exception would be the case in which the creation of a PA involves the addition of component variants to those parts of the architecture which interact with the sensitivity points. In the example given above, consider adding a third component to periodically receive exception messages from both components. If such notification messages to this third component are not similarly encrypted, the security of the system may be jeopardized.

An area of the architecture which is a sensitivity point and which contains at least one variation point, will be referred to as an *evolvability point*. Such variation/evolvability points deserve special treatment, as they have the potential to alter (and, possibly, damage) the quality of the architecture(s). In order to defuse that potential, each evolvability point in the CA is accompanied by suitable guidelines to constrain or guide subsequent PA design decisions and conformance checking. Thus, the developers are warned against making design

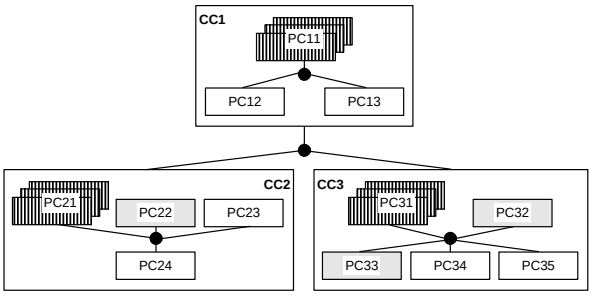

Figure 1. Example product line architecture adapted from [7]. Unshaded boxes represent mandatory components; vertically striped boxes represent alternative components; shaded boxes represent optional components.

decisions in a PA that could invalidate the quality goals already identified in the CA.

As the CA changes, the evolvability constraints (or quality attributes conformance constraints) are updated accordingly to guide future design of the PAs. The evolvability points also help simplify maintenance because the architects would be rightly guided to those critical design decisions that controls quality attribute responses.

As an example, consider the architecture shown in Fig. 1, which is made up of three complex (or composite) components CC1, CC2, and CC3. Each of these components is in turn made up of a number of primitive components. In the product line approach, those primitive components can be identified as mandatory (or common), optional, or alternative. Mandatory components, by definition, are fully specified in the CA and are always present in any PA. Optional components are underspecified as variation points in the CA; they can become fully specified as components (or variants) in a given PA, or they will not be present at all. Finally, alternative components are underspecified as variation points in the CA but must become fully specified into some component (or variant) in the PA.

In the definition of this architecture, design decisions that interact with one or more quality attributes (i.e., sensitivity points) are assumed to be located in some of the components. Let's assume that performance and availability are the two quality attribute goals of the highest priority. We shall consider two scenarios in relation to the architecture illustrated in Fig. 1: in the first scenario, the architecture is taken to be a product architecture (PA), while in the second, it is taken to be the core architecture (CA).

Scenario 1: architecture is a PA

If the architecture in Fig. 1 is a product architecture, then the shaded and unshaded boxes are fully specified architectural components (i.e., primitive components). In this scenario, we will consider two possibilities concerning the nature of the sensitivity points.

In one case, let the sensitivity points be located in the mandatory components whose design decisions are preset in the CA. For example, the sensitivity interacting with performance is localized in PC23, while that of availability is localized in PC24. Since both sensitivity points are localized in mandatory components, each individual PA inherits those sensitivity points intact. With them, performance and availability qualities are inherited from the CA. As a result, the availability and performance quality will always be met in this PA. In fact, every product built from that CA is guaranteed to provide the preset quality responses for performance and availability.

Alternatively, one or both quality attributes may be localized in an optional or alternative component. Let us assume that the performance quality of this PA is determined through the appropriate design decisions of CC2. Further, assume those design decisions are jointly localized in components PC24 (mandatory) and PC21 (alternative). The design decisions of PC24 are determined during the CA definition, while those of PC21 are determined in this particular PA definition. If the correct guarantees for performance are provided through PC24, but not through PC21, the desired performance response may not be guaranteed. To avoid this, the PA must correctly specialize PC21 from its variation point definition in the CA; to this end, relevant design decisions need to be guided or constrained in an appropriate way, as described below.

Scenario 2: architecture is the CA

In this second scenario, let us assume that the architecture in Fig. 1 is the CA, in which case only the white boxes are fully specified, while the shaded and striped ones correspond to variation points of either optional or alternative type. As in the previous scenario, there are two possible cases to consider.

If all the sensitivity points in this architecture are located in mandatory components (which should be the goal of every product line design), then the CA design decisions will address the common quality of all products.

However, the above case is not always what is obtained in reality. Oftentimes, there are two or more sensitivity point localized in both areas that has been fully specified (e.g., mandatory components) and areas that are not fully specified (variation point). The architects can only design to fulfill the quality goal of the mandatory component and expect product architects to fulfill their part in designing the variants for the appropriate quality response. If the teams are different, this may be hard to do without duplication of efforts.

To ensure conformance of the PA design decisions to those of the CA, in order to fulfill a common quality goal, an evolvability point and evolvability constraint pair are needed. It is not every variation point in the CA that is an evolvability point, but only those that interact with the sensitivity point. The designers of the CA will accompany such evolvability points with constraints/guidelines to help product architects in their work.

Evolvability constraint is a statement about an evolvability point that guides product architecture creation in order to fulfill desired quality goals. Just like every other form of constraints, it may be described using the syntax and semantics of an ADL or other constraint language. The constraint may restrict variant components in their interaction protocol, internal states, architectural styles, implementation or usage [9] in order to fulfill some quality goals.

The combined use of evolvability point and evolvability constraints ensures that PAs remains in conformance with the CA. The following is a description of an evolvability point (EP) and its corresponding evolvability constraint (EC), as defined in a recent case study of a product line called btLine, in the domain of mobile and electronic payment systems.

EP: The response time of the btLine product to tasks delegated to it is dependent on whether it is interfaced directly to the legacy and back office systems of its host organisation or not. The fact that design decision on product integration varies from product to product makes it an evolvability point.

EC: To enhance response time for transaction involving a product, external data request from within the product (e.g., balance of a customer account in the host banking system) must not involve complicated and time-consuming queries. Alternatively, an external integration mechanism may be deployed to synchronize account details between the bank systems and their local btLine product; of course with guidance from the btLine team. Better still, outbound request from a btLine product to external systems may be routed to a low-traffic data source or business component for improved response time.

4. Conclusion and Open Issues

We highlight the problem context and the challenges of ensuring quality attributes conformance between a product line common CA and its product PAs. Subsequently, we described a technique for implementing this form of conformance during product development and maintenance. The technique focuses on identifying variation points that interact with sensitivity points. Those points, refered to as evolvability points, are accompanied with suitable guidelines and/or constraints. The constraints inhibit any PA design decisions from degrading the preset quality attributes' responses of the CA. Adhering to the constraints and guidelines would ensure that the quality attributes of the PA are in conformance with those of the CA.

The main contributions of this approach include its architecture-centric focus for reasoning about quality attributes conformance of the product architectures to the CA; and systematic use of variation points to constrain product architectures from deviating from the preset qualities of the CA. Both of these should facilitate understanding of the interactions, conflicts, and tradeoffs between quality attributes of different forms of architecture encountered in product line development.

Much of the issues relating to quality attributes conformance between the CA and the PAs are still open. First and foremost, considerable advances have been made regarding architecture recovery from existing systems – but extraction of CA and PAs from such systems is still an open area for research.

Second, there is need for characterizing those areas of the CA that do not feature any variation points, but that have the potential of determining qualities both in the CA and the PAs.

Other open questions include: What approach can be used to resolve quality attributes conflicts between the CA and PA? How responsive is the current result to product line development in the evolutionary approach involving reverse engineering or reengineering? What is the impact of the CA evolving in terms of functionality and quality on the quality responses of the product architectures? How can software product line specialists utilize the result of the characterizations of conformance checks between a product line's CA and PAs for checking conformance of the code-dependent (as-built) architecture to the documented (as-designed) PAs?

Finally, while tool support is always a plus, the exact details of support for quality conformance checking and traceability in a product line context have yet to be worked out.

Some of these issues will be addressed in our future research.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, 2nd edition, 2002.

[2] J. Bosch. *Design & Use of Software Architectures*. Addison-Wesley, Harlow, England, 2000.

[3] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. *ICSR '98*, p. 308, Washington, DC, USA, 1998.

[4] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures – Methods and Case Studies*. Addison-Wesley, Reading, MA, 2002.

[5] P. Clements and L. Northrop. *Software Product Lines*. Addison-Wesley, Reading, MA, 2002.

[6] P. Clements and L. Northrop. *A Framework for Software Product Line Practice Version 4.2*. Software Engineering Institute, 2005.

[7] E. Dincel, N. Medvidovic, and A. van der Hoek. Measuring product line architectures. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 346–352, London, UK, 2002.

[8] R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. CMU SEI Technical Note CMU/SEI-2000-TR-004, ADA382629, Software Engineering Institute, Pittsburgh, PA, 2000.

[9] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. *ESEC'97/FSE-5*, pp. 60–76, Zurich, Switzerland, 1997.

[10] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[12] C. Stoermer, L. O'Brien, and C. Verhoef. Practice patterns for architecture reconstruction. *WCRE '02*, p. 151, Washington, DC, USA, 2002.

[13] C. Stoermer, L. O'Brien, and C. Verhoef. Moving towards quality attribute driven software architecture reconstruction. *WCRE '03*, p. 46, Washington, DC, USA, 2003.

[14] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *J. Syst. Softw.*, 66(3):225–239, 2003.