Proposing a new method for non-functional requirements analysis in software product line

Mohammadreza Mohammadrezaei
Computer department, ramhormoz branch, Islamic Azad University, Ramhormoz, Iran
mohammadrezaei@iauahvaz.ac.ir

Abstract
Software product line development has long been considered in the life cycle of engineering domain and application engineering. Product line requirements analysis is an important phase in the domain engineering. Analysis of the requirements in this area, like individual systems, involves functional and non-functional requirements. In the area of functional requirements analysis of the product line, some work is done, and several methods have been proposed. However non-functional requirements analysis has been neglected. This study has offered a comprehensive method for non-functional requirements analysis of software product line by identifying needs of a suitable method, and it has tried to overcome the deficiencies of existing approaches and methods. CNFRs analysis method has created a comprehensive analysis of discovery of implicit interdependencies between requirements. The main output of provided method is a completed feature model which involves functional and non-functional requirements. Therefore, the features model and the relationship rules table are the artifacts of the domain analysis in CNFRs product line. The proposed method is evaluated with a case study. The case study results CNFRs method and compare it with previous based on the results of the metrics indicating an improvement in the suggested method, including higher safety of real products derived from the features model, greater integration in functional and non-functional requirements in the features model, greater accuracy in presenting real searchable products, and greater avoidance from problems resulting from the interaction of features. Using CNFRs method, developers analyze all product line requirements, and the method artifact model presents real searchable products with more accuracy than previous methods.

Keywords: software product line, domain engineering, non-functional requirements analysis

1. Introduction
Software product lines (SPLs) are used to generate a variety of related programs that are tailored to specific use cases [1], [2]. By reusing assets in different variants (i.e., programs), SPLs achieve a rapid product deployment and reduce costs. To generate a tailor-made variant, a stakeholder selects the features (functionality) according to her requirements. This way, users can avoid an overhead in functionality for a variant such as a full featured database system in an embedded system. However, tailoring the variant regarding functionality alone is often not sufficient. In practice, non-functional properties (NFP) gain momentum. Power awareness, as a non-functional property, is a promising research field [3], [4]. In Green IT, alternative implementations of special algorithms such as sorting [5], are developed to reduce power consumption. Nonfunctional properties are especially important in the field of resource-constrained systems in which binary size and memory consumption are limiting factors. These heterogeneous non-functional requirements often lead to a redevelopment of already existing functionality. Software product line engineering has been proven to be useful to tailor a variant for functional and non-functional requirements without the negative impact of redeveloping large parts of a software. Variability provided by an SPL should enable the generation of variants that are equal with respect to functionally but differ in their non-functional properties. To this end, SPLs should provide alternative implementations of the same functionality that are optimized for specific NFPs. For instance, by implementing a feature in different ways, e.g., a performance optimized variant and a footprint optimized variant of a feature. These implementations introduce new variation points in the SPL to be exploited during the configuration process [6], [7]. Our aim is to provide differently optimized variants of an SPL based on a single architecture which is different from other approaches [8], [9]. The positive effect of having a single architecture is that
software evolution and maintainability is easier. While the general idea of optimizing NFPs includes also the selection of alternative implementations, in this paper, we focus on refactorings. Refactorings are changes in the structure of source code without altering the program semantics [10]. We categorize suitable refactorings according to their influence on non-functional properties in Section III-0a. For example, refactoring Inline Method can increase the performance, however, it might also have a negative effect on binary size. Based on our categorization, a user chooses suitable refactorings that optimize the source code during the configuration process. Each refactoring is defined in a single module, called refactoring feature module (RFM) [11], and is applied based on the configuration process. This way we can change a variant according its non-functional properties independently of the compiler or programming language, e.g., by decreasing the binary size by selecting the Pull up Method refactoring or by increasing the performance through Method Inlining. We make the following contributions: (a) We present an overview of tasks that are required to optimize NFP of SPL variants. (b) We show a concrete optimization technique based on refactorings including a proof of concept.

2. Software Product Line Scenario

In SPL development, we differentiate between domain engineering and application engineering. A domain engineer analyzes the functional and non-functional requirements that are important for an entire domain. This is in contrast to conventional software development in which concrete requirements are defined and are usually known before development. These requirements address the whole spectrum of possible program variants and may be contradicting. For example, a DBMS SPL can contain features for in-memory and persistent storage. Although both features have contradicting goals, they both are useful for specific scenarios. That is, developers implement alternative features to satisfy different and even incompatible goals. After domain analysis, developers and domain engineers design and implement a reference architecture for the SPL. Hence, typically almost the whole implementation work is done in the domain engineering phase. For each product, application engineering starts with the requirements analysis of a concrete application scenario. After this second requirement engineering phase, an application engineer selects features to satisfy her requirements. If requirements cannot be satisfied, new features or alternative implementations have to be developed. Developing a DBMS SPL would start by analyzing the database domain. Domain engineers identify common and variable functionality, such as data structures, search indexes, encryption mechanisms, transaction support, and logging. A feature model is used to document the features of an SPL including their dependencies.

3- Measuring Non-functional Properties

Many but not all non-functional properties can be measured. Measurement theory defines multiple scales, such as ordinal, interval, and ratio. We measure non-functional properties that can be described with a metric scale for which a stakeholder can define a suitable metric. For example, we can define footprint and performance (using a benchmark that outputs performed transactions per second) as properties to be measured for Berkeley DB. By contrast, it would be difficult to define a metric to measure user-friendliness. Hence, we differentiate between quantifiable and qualitative properties, which we explain In SPL engineering, developers face the problem that requirements of concrete customers are specified after SPL development. That is, SPL vendors have to consider a spectrum of non-functional properties during development. Typically, it is not known without exhaustive measurements which implications a feature selection has on certain non-functional properties. We illustrate the relationship between three features of Berkeley DB and the four non-functional properties footprint, performance, reliability, and security. Often, a feature affects multiple non-functional properties, for example, feature Replication of Berkeley DB increases the binary size by 89KB. However, this information is not known until we have actually measured it. Other non-functional properties such as reliability cannot be measured at all. Their influence can only be described qualitatively, rather than quantitatively. For example, we may also need domain knowledge to somehow express the influence of a feature on such a property. Even worse, also a certain feature combination has an influence on nonfunctional properties. We show program variants with different feature combinations. The variant that includes both features Replication and Cryptography has an unexpected behavior. We obtain a decrement in performance although, when measuring a variant with only a single feature, there is no performance decrement compared to the base variant.
Moreover, based on the feature’s footprint, we would expect that the variant has a size of 448KB rather than 480 KB. The observed difference is caused by feature interactions of both features at the source-code level. Another example is SQLite. SQLite is a customizable DBMS SPL deployed on over 500 million systems (SQLite.org, 2010). Although it targets embedded systems and thus has a small footprint, the developers provide further configuration options to reduce the size of the compiled DBMS. However, they can neither provide values to which degree a deactivated feature saves binary size nor what influence a deactivation has on other non-functional properties. The website states only the library size can be less than 300KiB, depending on compiler optimization settings.”. and ”If optional features are omitted, the size of the SQLite library can be reduced below 180KiB.” Often, a customer needs more exact information than ”less than” or ”can be reduced below”. Hence, to find a feature set for a specific footprint limit, a customer would need to measure the binary sizes of many variants. Considering the fact that 88 features are optional and can be arbitrarily configured, there are 288 different variants. Measuring all variants would take longer than the time the universe exists. Obviously, a customer cannot find the optimal variant with a brute force approach.

Fig: Relationship between non-functional properties and feature interactions

4-conclusion

In this paper, we address the problems of measuring non-functional properties and finding the optimal variant for given non-functional requirements. With SPL Conqueror, we present a holistic approach for the whole variant-derivation process

Acknowledgements

Funding support of this research was provided by the ramhormoz branch, Islamic Azad University, Ramhormoz,khoozestan, Iran.

References