Optimizing Non-functional Properties of Software Product Lines by means of Refactorings

Norbert Siegmund, Martin Kuhlemann, Mario Pukall
Department of Computer Science
University of Magdeburg
Magdeburg, Germany
Email: {nsiegmun,mkuhlema,pukall}@ovgu.de

Sven Apel
Department of Informatics and Mathematics
University of Passau
Passau, Germany
Email: apel@uni-passau.de

***Abstract*—Today, software product line engineering concentrates on tailoring the functionality of programs. However, we and others observed an increasing interest in non-functional properties of products. For example, performance, power awareness, maintainability, and resource consumption are important non-functional properties in software development. Current product line techniques have the potential to flexibly optimize non-functional properties. In this paper, we present our vision of optimizing non-functional properties in software product lines. We show how such an optimization can be achieved using refactorings and present first results of a case study.**

***Index Terms*—software product lines; non-functional properties; product derivation;**

I. 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. Non-functional 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.

II. Vision

The configuration of an SPL is guided by a feature model. A feature model is created by a domain engineer to define the features of the SPL [12], [13]. Using a feature model as the basis for the variant configuration, it should be possible to optimize NFP of the desired variant. Before a variant can be optimized for an NFP, we have to provide mechanisms that allow a user to measure and configure the property. This is a non-trivial task because the properties are heterogeneous in their nature. For example, even though we can easily measure the binary size of individual features and can aggregate these values for a specific variant, it is difficult to measure Security and Reliability. The reason is that it is difficult to define a metric for those properties and even harder to find a meaningful aggregation function in order to compare different variants.

There are some models in literature that classify non-functional properties across the software life cycle [14], [15]. Whereas these classifications provide a good overview of possible non-functional properties, they are insufficient for our needs. In order to enable the configuration of such properties, we need a new classification in which each class provides different measurement techniques and configuration mechanism. In [6], we presented three classes of non-functional properties. The first class, called *Directly Assigned Properties*, contains the properties that cannot be quantified. This is the case for Security or Reliability, for which a domain expert cannot define comparable values. These properties can be seen as non-functional features because their configuration completely correspond to the common feature selection. However, not every NFP can be represented as a single non-functional feature, e.g., the footprint of features.The missing quantification affects the optimization and configuration because we have no comparable values, we cannot define optimizations. However, we can hint the user to features that have a positive or negative impact on the required NFP; we directly assign the property to corresponding features in the feature model.

For the category *Inferred Properties*, we can measure the influence of a feature regarding a property, e.g., the influence of a feature on the binary size of a variant. Using different user-defined metrics, we can measure or estimate values for single features and annotate these values to the corresponding feature in the feature model. This allows us to compute in advance the aggregated value for a variant which, for example, can be used to compute the influence of differently selected alternative implementations of a single feature. As a result, a user can define objective functions for desired non-functional properties and an optimizer can then select the best configuration. For example, a user might want to minimize the Power Consumption and keep the footprint below 200 KBytes. Considering V is the set of all valid variants of an SPL and *Footprint* represents the binary size of a variant whereas *Power* the power consumption respectively. Then she could define the following objective function to derive the optimal variant v_{opt}:

$$v_{opt} = v_i \Leftrightarrow v_i(Power) \leq v_j(Power)$$
$$\wedge v_i(Footprint) < 200KB \wedge v_{i,j} \in V \wedge i,j \in \mathbb{N} \wedge i \neq j$$

The last category covers *Runtime Properties* which emerge only in a running variant. This makes these NFPs difficult to measure because we first have to configure, compile, and run a candidate variant in order measure its NFPs. Prominent examples of *Runtime Properties* are Performance and Memory Consumption. Due to the measurement effort, we propose the configuration process to incrementally reduce the number of candidate variants so that only a few variants have to be executed [6].

A. *Configuration of Non-functional Properties*

The configuration of a variant begins with a user's feature selection. This step defines the functionality a variant has to provide. During the next step, a stakeholder selects the features that improve a non-functional property (category *Directly Assigned Properties*). Although, such a selection is only an extension of the feature selection phase, it is important for the optimization of non-quantifiable properties. A tool can support the stakeholder in this phase by highlighting and grouping suitable features. Thus, the difference between step 1 and 2 is the reason for the feature selection, i.e., a user selects features for purpose of required functionality in step 1 and in step 2 for NFP-optimizing features. During the subsequent step, a user defines constraints regarding non-functional properties, e.g., a variant must not exceed a footprint more than 200 KBytes. These constraints reduce the number of possible acceptable variants, which is important for the measurement of *Runtime Properties*. Afterwards, a user can define an objective function for optimizing a certain property. Based on such a function, the respectively best implementations are automatically selected for runtime measurements.

Whereas in current approaches, the configuration of a variant's functionality constitutes the end of optimization, we apply further optimizations through refactorings. We apply refactorings that have an impact on non-functional properties but do not alter the functionality. These refactorings, defined by a developer or automatically generated during the configuration, can be seen as additional configuration options of the SPL for improving desired NFPs. User defined refactorings can then be selected in step 2. When a refactoring is not part of the SPL but would contribute to the desired NFP, we generate an RFM accordingly. In Section III, we present in detail which refactorings can be automatically applied to improve a certain variant and where manually defined refactorings should be used.

To realize our vision, we have to extend the common SPL development process [13]. In Figure 1, we provide an overview[1] of such an enriched development process. In the upper part, the usual SPL development process is shown. It starts with the domain analysis in which features are identified and the granularity of the variability is specified. Developers continue to implement the defined features. Different techniques are possible which also impact the configuration and optimization of NFPs.

SPLs are often implemented with preprocessor statements like *#IFDEFS* in C and C++ or with components. Also new techniques, e.g., *aspect-oriented programming* [16] and *feature-oriented programming (FOP)* [17], [18] can be used. FOP is a technique to encapsulate feature code in distinct feature modules (FM) (see Figure 1). By selecting features in the product derivation step, the corresponding feature modules are composed to create the desired product. We propose to add a new concurrent process which focus on non-functional

[1]Note, that the SPL development is usually separated in domain and application engineering.

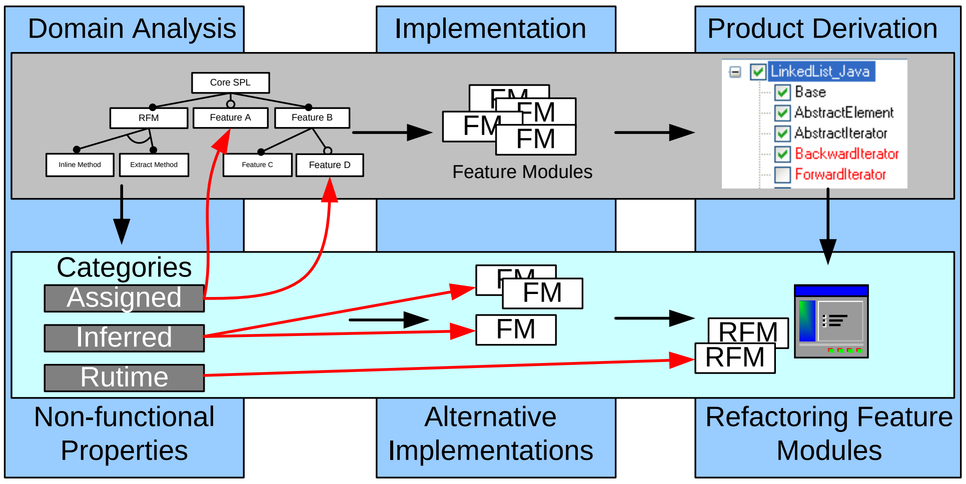


Fig. 1. SPL development including the optimization of non-functional properties.

properties. An additional development team is responsible to identify important properties for the domain, to develop alternative implementations, and to define refactorings (inside RFMs) for these properties. The development of such additional feature models is separated from the common implementation of functionality, e.g., the implementation of alternatives for a new customer with specific needs in a NFP. This way, the development of functionality is independent from the optimization of NFPs. With such an enriched product line engineering approach, we can improve the maintainability of the SPL's source code (separated feature modules for different NFPs). Additionally, we can decrease time-to-market, because one engineering team focuses on the functionality while another team is responsible for tailoring the SPL variants regarding NFP requirements or target systems. As the presented additional process does not affect the common SPL development, existing SPLs can adapt this methodology.

B. *Measurement Framework*

Due to the large area in which SPLs can be applied, metrics for the same NFP can often not be reused across different SPLs. Commonly, a domain has a strict specification for NFPs and the corresponding metrics. For example, Performance metrics in the area of database management systems are often evaluated with benchmarks whereas the same property, e.g., in SOA, is often expressed in terms of system response time. The metrics and corresponding optimizations, we have already implemented and tested, are specific to the actual domain, but we need a general methodology that allows users to integrate their own specific metrics and aggregation functions.

Based on this insight, we claim that a *framework for measuring and aggregating NFPs* is required in which SPL developers can plug in their domain specific metrics. The framework should provide basic functions that measures individual features (*Inferred Property*) or variants (*Runtime Property*) based on these metrics. Inside a plugin, a developer can use an existing tool (e.g., for measuring the cyclomatic complexity), of a program or feature. The framework could pass automatically each feature into the plugin which in turn can pass the feature to the desired measurement tool. The results can then be annotated to the respective feature in a feature model. To aggregated the values of different features for a variant the plugin must also define an aggregation function, e.g., for cyclomatic complexity it might be the "maximum". During the configuration, a user can now define a constraint to keep the complexity below a certain number. Using the aggregation function, variants that cannot fulfill this requirement are removed. We use a first implementation of this framework for our refactorings in order to generate refactorings based on user-defined metrics.

We have given an overview how non-functional properties are related to SPLs. We described a classification to highlight that the optimization and configuration of non-functional properties require different methodologies and reflected some optimization possibilities. In the following section, we present a new technique that does not affect the architecture of a variant, i.e., the selected features and feature modules. Therefore, this technique can be used on top of already existing approaches and provides further opportunities to tailor a variant according non-functional properties.

III. Optimizing NFPs with Refactorings

Refactorings alter the structure of source code without changing the application behavior [10]. Depending on the type of refactoring, different non-functional properties can be affected. For example, the *Inline Method* refactoring can improve the execution time because it replaces the method call with the body of the called method. A recent study has shown that removing delegation can improve the performance of a program by 50% [19].

Applying refactorings provides new optimization possibilities to a user. We want to exploit these possibilities and select refactorings according to the given optimization goals. Besides such an optimization, a further advantage of this approach is compiler and platform independence because the refactorings are applied to the source code of a variant. Furthermore, developers are not forced to implement their SPL in a particular way, e.g., coding guidelines defined by a customer can be realized after development. The developers can define the refactorings seperately and keep the core architecture of the SPL stable for maintenance. Besides the definition of refactorings by the developers, it should be possible to generate refactorings according to a certain metric. For example, if a user wants to optimize the performance with a given metric for method inlining, a tool should automatically select suitable refactorings or generate them if they do not already exist in the SPL.

Given these requirements, we developed a technique to define and reuse refactorings in SPLs like feature modules. *Refactoring feature modules (RFMs)* integrate refactorings with feature-oriented programming (FOP) [11]. The goal of both techniques, RFM and FOP, is to successively transform a base program. Whereas modules in FOP transform the functionality of the base program, RFMs transform the structure of the base program. Once defined or generated, RFMs become user-selectable features in a feature model.

The main focus of RFMs so far was in program integration [11]. RFMs are defined by the programmer as part of the product line design and selected by the user in order to overcome incompatible structure of a variant with an external application. For instance, to reuse an existing library in a client application, classes of this library might need to be renamed in order to be compatible [11].

We use RFMs to manipulate non-functional properties. While RFMs can be defined and selected manually for integration purposes, their manual definition and selection becomes unfeasible when they should adapt NFPs. The reason is that, to improve the NFP Performance, potentially hundreds or thousands of RFMs must be defined and selected, of which each inlines one method (*Inline Method* refactoring).

a) Selection of Refactorings: There are numbers of refactorings described in literature. Our study builds on the refactorings defined by Martin Fowler [10]. In a first step, we analyzed the refactorings and came up with an approximated influence on NFPs for every refactoring. The analysis based on the known influence of different program executions when applying different refactorings, e.g., removing setter methods can result in less compiler instructions and thus may improve the performance. We plan to evaluate in a detailed case study the influence of the most common refactorings. The results are given in Table 1. Note, that applying a refactoring *can* improve or degrade a property but does not have to. We need additional metrics, e.g., those for method inlinings used in compilers, to achieve the desired effect. In the following, we describe the results of our analysis exemplary for some NFP:

- **Performance.** To reduce the execution time for method calls a programmer can apply refactorings like *Inline Method, Inline Class, Remove Middleman*. This is done by replacing a call with the called method's body. The method call is removed but the same actions happen as before, so performance is improved. However, when methods grow too large, this results in cache mismatches of the processor [20]. These mismatches arise because the method is too large to fit in the cache completely and instead must be reloaded, which increases the execution time. To overcome such problems different metrics exist, e.g., for compilers to achieve the best performance.
- **Footprint.** The footprint of an application is the sum of the footprint of each compiled file. For Java, we measure the class files that contain intermediate byte code. By removing (setting) methods or code clones (e.g., by transforming members to parent classes using *Pull up Field* or *Pull up Method* refactorings), the footprint can be reduced. Note, that these refactorings often have only a small influence to shrink the binary size. In contrast, for example inlining methods in multiple other methods may result in an expanded footprint. This must be considered when footprint constraints are defined.
- **Coding styles.** Coding styles are important if products are sold as source code libraries to multiple customers where each customer has its own styling guideline. There are different tools that check the validity of code against coding rules, e.g., Checkstyle[2]. For such rules, refactorings (e.g., *Extract Method* or *Rename Method*) can be automatically generated and applied on demand. Although, this approach might be a possible way to pass a program validator, the maintainability for developers will rather be decreased because of generated names for methods and variables. A possible solution are developer-defined RFMs that are selected on demand. With RFMs functional requirements of an SPL are separated from non-functional requirements, e.g., different code guidelines of different customers. A variant can be quickly adapted to fit the needs of new customers when existing RFMs are reused.

We have collected *some* possible use cases for optimizations using refactorings. The suitability of each refactoring depends on the program and on the quality of the metric that defines which refactorings have to be used. Both types of usage, generation and manual definitions are required. Automatically generated refactorings are useful if a high number of refactorings is necessary. Manually implemented RFMs should be used if developers knowledge is necessary and soft properties like "Readability" must be improved.

IV. Proof of Concept

Currently, we are developing a tool which tackles the following requirements: (a) configuration of a variant regarding functionality and *Directly Assigned Properties*, (b) definition of objective functions for NFPs, (c) automated selection of optimal implementations regarding NFPs, and (d) the definition, selection, and appliance of RFMs. The NFP optimizer supports SPLs implemented with feature-oriented programming (FeatureC++ [21] and JAK [18]). In [6], we have shown a possible solution to compute an optimal selection of alternative

[2]`http://checkstyle.sourceforge.net/`

Non-functional property	Improve	Decrease
Performance	Inline Method, Inline Class, Remove Middleman, Remove Setting Method, Replace Delegation with Inheritance, Replace Temp with Query, Inline Temp	Encapsulate Field, Extract Class, Extract Method, Form Template, Introduce Assertion Method, Hide Delegate, Replace Inheritance with Delegation, Self Encapsulate Field, Change Unidirectional to Bidirectional, Decompose Conditional
Footprint	Collapse Hierarchy, Pull up Constructor Body, Pull up Field, Pull up Method, Remove Middleman, Remove Setting Method	Decompose Conditional, Encapsulate Field, Extract Class, Extract Interface, Hide Delegate, Inline class, Inline Method, Inline Temp, Introduce Assertion, Introduce Explaining Variable, Push down Field, Push down Method, Remove Assignments to Parameters, Self Encapsulate Field
Styling Guidlines and Code Metrics	Extract Method, Replace Conditional with Polymorphism, Replace nested Conditional with Guard Clauses, Extract Method, Create Template Method, Consolidate Duplicate Conditional Fragments	Inline Method, Replace Exception with Test, Inline Method
Readability	Extract Class, Extract Subclass, Extract Superclass, Inline Method	Collapse Hierarchy, Consolidate Conditional Expression, Decompose Conditional, Encapsulate Field, Extract Method, Hide Delegate, Inline Class
Object Size	Inline Temp	

TABLE I
OVERVIEW OF REFACTORINGS AND THEIR INFLUENCE ON NON-FUNCTIONAL PROPERTIES.

implementations for the NFPs Cyclomatic Complexity, Footprint, and Performance without using refactorings. We extend the tool to support our approach by using RFMs. After the derivation of a variant, the user has now the opportunity to further improve a certain non-functional property. Currently, we only support Performance, however, in future we will provide optimizations for additional NFPs.

After a variant is configured, we use JastAdd[3] to analyze the abstract syntax tree of the composed variant. In particular, we search for methods where refactorings can be applied in order to improve performance. We additionally analyze where inlinings might be reasonable, e.g., method calls in loops. However, we exclude methods that are polymorphic and recursive, as inlining those methods is not yet implemented. The output of this analysis is a list of candidate methods (see left part of Figure 2).

Such a listing is interesting for stakeholders with programming skills because methods can be manually marked for inlining where a positive effect can be foreseen. This is the same as the *inline* keyword in C or C++.[4] However, normally a user defines a metric that defines when methods should be inlined, see top right in Figure 2. This enables a compiler-independent definition of an inlining metric and has therefore effects independent of the underlying system. After defining refactorings beneficial for NFPs of SPL variants, we compute and generate (or reuse existing) RFMs to reach the optimization goals. The selected refactorings are shown to the user (right part of Figure 2). Subsequently, the tool applies the RFMs and compiles the new variant. The performance of the synthesized product is measured and compared to the performance of the product without refactorings. Although, we currently support only inline method refactorings, the results in Figure 3 show that we achieve performance improvements for certain cases (when we applied RFMs, we never produced an inferior result compared to the non-optimized variant).

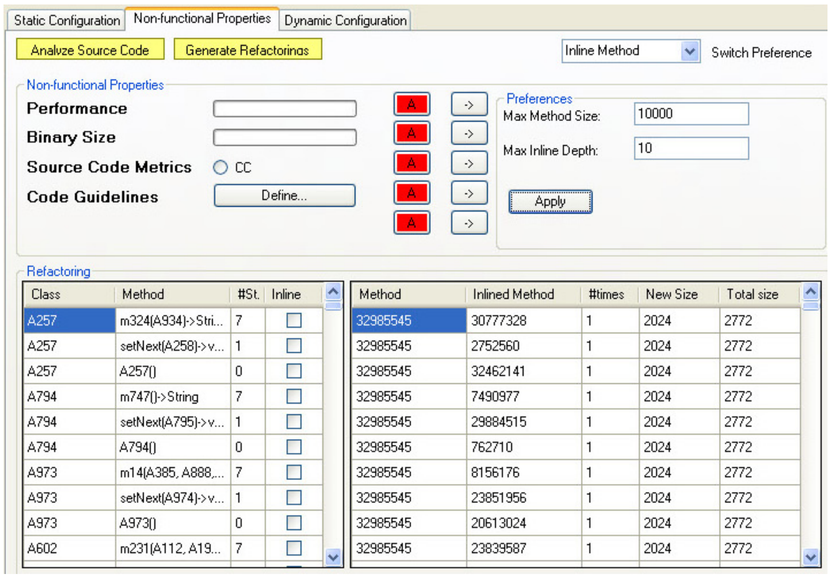
b) Case Study: We present our first results for the optimization of NFP using RFMs. For our case study, we used the micro benchmark presented in [19], which implements a delegation chain.

After the analysis, the system came up with 980 possible methods for inlining. We defined a metric to restrict the method size to 1000 statements and the maximal inline depth to 10. Applying more sophisticated metrics that cover more setup possibilities, e.g., preferred inlining in loops, is left for future work. Our tool computed 120 Inline Method refactorings based on our metric. As no RFMs were present in the SPL before, our tool generated all of them. After applying these RFMs, we measured the performance 10.000 times to get significant data. The results are given in Figure 3. The X-axes shows the intervals for time needed to pass the performance test. For each interval, we counted the number of executions which are depicted in the Y-axis. In the result, the execution times were significantly better with our optimizations than without. We found that refactorings can be successfully applied to an already optimized variant to further improve an NFP. The refactorings have also an influence on the footprint property. In the unoptimized version the sum of all class files requires 705,569 bytes whereas the variant optimized using refactorings consumes 833,021 bytes. We see our assumptions according the influence of refactorings to NFPs approved and expect additional optimization benefits if more refcatorings are supported.

V. Related Work

Product derivation tools try to guide the user through the whole derivation process. There are commercial tools like pure::variants [22] and Gears [23] that contain mechanisms to maintain and develop an SPL as well as scientific tools [24], [25], [26]. These tools allow developers to create feature models and guide users through the configuration process with

[3]http://jastadd.org
[4]The *inline* keyword is used in front of method declarations to force the compiler to inline the method.



Inline Method

Max Method Size: 10000

Max Inline Depth: 10

Class	Method	#St.	Inline
A257	m324(A934)->Stri...	7	☐
A257	setNext(A258)->v...	1	☐
A257	A257()	0	☐
A794	m747()->String	7	☐
A794	setNext(A795)->v...	1	☐
A794	A794()	0	☐
A973	m14(A385, A888,...	7	☐
A973	setNext(A974)->v...	1	☐
A973	A973()	0	☐
A602	m231(A112, A19...	7	☐

Method	Inlined Method	#times	New Size	Total size
32985545	30777328	1	2024	2772
32985545	2752560	1	2024	2772
32985545	32462141	1	2024	2772
32985545	7490977	1	2024	2772
32985545	29884515	1	2024	2772
32985545	762710	1	2024	2772
32985545	8156176	1	2024	2772
32985545	23851956	1	2024	2772
32985545	20613024	1	2024	2772
32985545	23839587	1	2024	2772

Fig. 2. Generating and applying refactorings in the NFP optimizer tool.

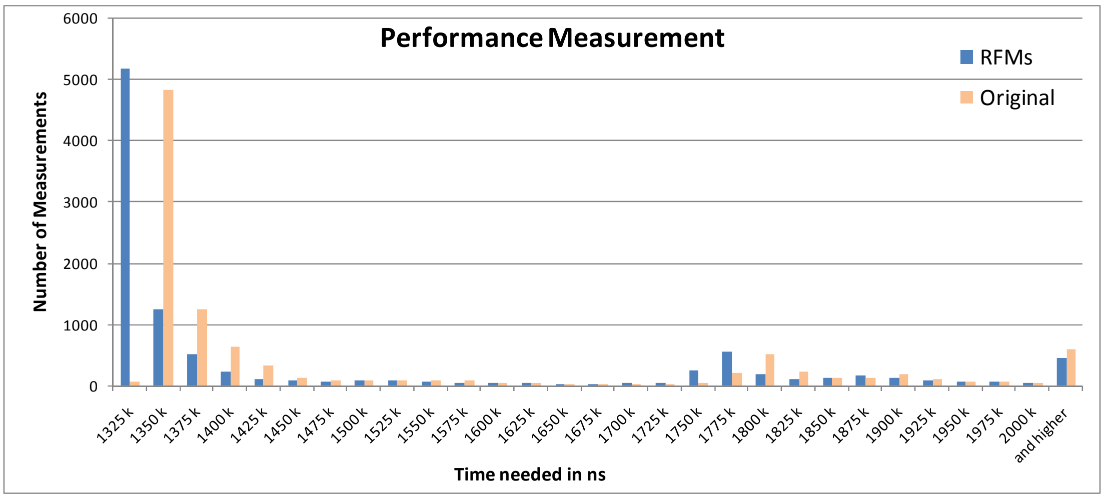


Fig. 3. Results for measuring a variant with and without RFMs.

special visualization techniques. Neither the measurement of non-functional properties nor the optimization for NFPs is supported for SPLs.

Benavides et al. [27], [28] presented a technique based on Constraint Satisfaction Problems (CSP) solvers to seek an optimal variant. The solver evaluates values attached to features in the feature model and then computes an optimal configuration for a small number of features. White et al. [7], [29] extended this approach to resolve resource constraints in the variant selection process. For large scale problems they propose a Filtered Cartesian Flattening to approximate a good variant. We see both approaches promising for an integration, e.g., for selecting optimal feature modules. However, we further provide an optimization technique based on RFMs and a framework to measure the values needed for optimization.

Other approaches use model-driven engineering techniques to generate different architectures optimized for certain quality attributes. In [8] components can be differently connected and interfaces are generated to obtain a valid program with modified quality attributes. Kim et al. [9] propose a framework, called DRAMA, which captures the requirements of users. Based on these requirements, different architecture styles, e.g., Layers or Model View Controller, can be applied to improve or degrade a NFP. The framework can also compare alternative implementations and chose the one with the best quality attributes. This approach is similar to our configuration of optimal implementations. We additionally include user selectable refactorings on source code level to further optimize a variant for NFPs and the measurement of NFPs.

Smith [30] uses correctness-preserving transformations in his tool *Kids* to improve the performance of a program. These transformations are similar to refactorings. Unlike Smith's transformations, RFMs can be seamlessly integrated into the SPL development process because RFMs represent reusable modules that can be described like features. Critchlow et al. [31] present an approach to use refactorings to change the architecture in order to improve certain quality attributes. They consider refactorings not at source code level but at architecture level to flexibly change the components architecture of a variant. We focus on refactorings that are applied to the source code and after a variant is created.

The Skoll project [32] targets on testing and measuring applications with large configuration spaces. The project tries to overcome the problem of having a huge amount of products by using a large number of users that share their computation power. For measuring runtime properties, this might be a suitable approach. However, the effort is very high and it will not scale with a large SPL. Our refactoring feature modules can be applied independent from the variant space.

Zhang et al. propose to use Bayesian Belief Network in order to analyze and predict non-functional properties based on the experience of the development of similar products and domain experts [33], [34]. The knowledge is captured and used for the development of new products to achieve suitable decisions for optimizing a certain non-functional property. This approach targets on architectural design and decisions during the design phase. Our approach can be applied after an SPL is developed and is therefor independent of architectural decisions.

VI. Conclusion

We presented our vision of optimizing non-functional properties (NFPs) of variants of software product lines (SPLs). We outlined difficulties in measuring and configuring NFPs and motivated the need for a framework that allows users to plugin their own metrics. Based on the measured values, our tool selects alternative implementations to optimize certain NFPs. We presented a new approach for optimization based on refactoring feature modules (RFMs). RFMs, defined by developers or automatically generated, are part of the SPL and can be selected like features to further improve NFPs. First results based on automatically generated *Method Inline* refactorings show that performance improvements can be achieved. In future work, we will support additional refactorings to further increase performance. In addition, we will support the optimization of other NFPs. Our long term goal is to provide a framework for which developers can implement plugins that optimize NFPs.

Acknowledgment

Norbert Siegmund is funded by the German Ministry of Education and Science (BMBF), project 01IM08003C. Mario Pukall is funded by German Research Foundation (DFG), project SA 465/31-2. Apels work is supported in part by DFG project #AP 206/2-1. The presented work is part of the ViERforES[5], RAMSES[6], and FeatureFoundation[7] projects.

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[2] C. W. Krueger, "New methods in software product line development," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2006, pp. 95–102.

[3] R. Jain, D. Molnar, and Z. Ramzan, "Towards understanding algorithmic factors affecting energy consumption: Switching complexity, randomness, and preliminary experiments," in *Proceedings of the Workshop on Foundations of Mobile Computing*. ACM Press, 2005, pp. 70–79.

[4] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the "best" sorting algorithm for optimal energy consumption," in *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, 2009, pp. 199–206.

[5] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "JouleSort: A balanced energy-efficiency benchmark," in *Proceedings of the 2007 International Conference on Management of Data*. ACM Press, 2007, pp. 365–376.

[6] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake, "Measuring non-functional properties in software product lines for product derivation," in *Proceedings of the 15th International Asia-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2008, pp. 187–194.

[7] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko, "Automating product-line variant selection for mobile devices," in *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2007, pp. 129–140.

[8] P. O. Rossel, D. Perovich, and M. C. Bastarrica, "Reuse of architectural knowledge in SPL development," in *Proceedings of the 11th International Conference on Software Reuse (ICSR)*. Springer-Verlag, 2009, pp. 191–200.

[9] J. Kim, S. Park, and V. Sugumaran, "Drama: A framework for domain requirements analysis and modeling architectures in software product lines," *Journal of Systems and Software*, vol. 81, no. 1, pp. 37 – 55, 2008.

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] M. Kuhlemann, D. Batory, and S. Apel, "Refactoring feature modules," in *Proceedings of the International Conference on Software Reuse*, 2009, pp. 106–115.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[13] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[14] J. A. Mccall, P. K. Richards, and G. F. Walters, "Factors in software quality. Volume I. Concepts and definitions of software quality." General Electric CO Sunnyvale California, Technical Report ADA049014, November 1977.

[15] "Software engineering - Product quality, Part 1: Quality model," 2001.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 1241. Springer Verlag, 1997, pp. 220–242.

[17] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 1241. Springer Verlag, 1997, pp. 419–443.

[18] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 6, pp. 355–371, 2004.

[19] S. Götz and M. Pukall, "On performance of delegation in Java," in *Proceedings of the International Workshop on Hot Topics in Software Upgrades*. ACM Press, 2009, pp. 1–6.

[5]http://vierfores.de

[6]http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/ramses/

[7]http://www.fosd.de/ff

[20] J. Dean and C. Chambers, "Towards better inlining decisions using inlining trials," in *Proceedings of the ACM conference on LISP and functional programming*. ACM, 1994, pp. 273–282.

[21] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming," in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ser. Lecture Notes in Computer Science, vol. 3676. Springer Verlag, Sep. 2005, pp. 125–140.

[22] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability Management with Feature Models," *Science of Computer Programming*, vol. 53, no. 3, pp. 333–352, 2004.

[23] Big Lever, "Gears," http://www.biglever.com.

[24] D. Streitferdt, M. Riebisch, and I. Philippow, "Details of formalized relations in feature models using OCL," in *International Conference on Engineering of Computer-Based Systems (ECBS)*. IEEE Computer Society, 2003, pp. 297–304.

[25] G. Botterweck, D. Nestor, A. Preußner, C. Cawley, and S. Thiel, "Towards supporting feature configuration by interactive visualization," in *Proceedings of Workshop on Visualisation in Software Product Line Engineering*, 2007, pp. 125–131.

[26] E. Cirilo, U. Kulesza, and C. P. de Lucena, "A product derivation tool based on model-driven techniques and annotations," *Journal of Universal Computer Science*, vol. 14, no. 8, pp. 1344–1367, 2008.

[27] D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "Automated reasoning on feature models," in *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, ser. Lecture Notes in Computer Science, vol. 3520. Springer Verlag, 2005, pp. 491–503.

[28] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés, "FAMA: Tooling a Framework for the Automated Analysis of Feature Models," in *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2007, pp. 129–134.

[29] J. White, B. Dougherty, and D. C. Schmidt, "Selecting highly optimal architectural feature sets with filtered cartesian flattening," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1268–1284, 2009.

[30] D. R. Smith, "Kids: A semiautomatic program development system," *IEEE Trans. Softw. Eng.*, vol. 16, no. 9, pp. 1024–1043, 1990.

[31] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek, "Refactoring product line architectures," in *International Workshop on Refactoring: Achievements, Challenges, and Effects*, 2003, pp. 23–26.

[32] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan, "Skoll: Distributed continuous quality assurance," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004, pp. 459–468.

[33] H. Zhang, S. Jarzabek, and B. Yang, "Quality prediction and assessment for product lines," in *Advanced Information Systems Engineering (CAiSE)*. Springer, 2003, pp. 681–695.

[34] H. Zhang and S. Jarzabek, "A bayesian network approach to rational architectural design," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, pp. 695–718, 2005.