Measuring Non-functional Properties in Software Product Lines for Product Derivation

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, and Gunter Saake
University of Magdeburg
39104 Magdeburg, Germany
{nsiegmun, rosenmue, mkuhlema, ckaestne, saake}@ovgu.de

Abstract

A software product line (SPL) enables stakeholders to derive different software products for a domain while providing a high degree of reuse of their code units. Software products are derived in a configuration process by composing different code units. The configuration process becomes complex if SPLs contain hundreds of features. In many cases, a stakeholder is not only interested in functional but also in non-functional properties of a desired product. Because SPLs can be used in different application scenarios alternative implementations of already existing functionality are developed to meet special non-functional requirements, like restricted binary size and performance guarantees. To enable these complex configurations we discuss and present techniques to measure non-functional properties of software modules and use these values to compute SPL configurations optimized to the users needs.

1 Introduction

A *software product line (SPL)* is developed to create a large number of related products by reusing a set of software artifacts called core assets or simply *code units* [10, 19]. These code units, once developed and tested, can be composed to derive different products with varying functionality. This decreases development effort and time-to-market while providing a high degree of reuse [16].

The functionality of an SPL is represented as features [18, 11]. *Product derivation* is the process of generating a tailor-made product of an SPL. It consists of (a) selecting features (functionality) according to stakeholder requirements, (b) checking the consistency of the selection, and (c) composing code units to generate the product. Even in small SPLs a manual selection is often a complex task [14]. If a user has additional non-functional requirements, e.g., binary size < 100 KByte, because the used device has restricted memory, the derivation process becomes even more complex and difficult. SPLs of industrial size can have thousands of features [32, 20] which makes a manual product derivation usually impossible. Current research addresses this problem by providing visualization techniques and special algorithms to reduce the complexity of the configuration process [33, 7, 35]. While these approaches are successful for functional requirements they do not address non-functional requirements.

Environments like embedded systems or large scale computing systems exhibit non-functional requirements like restricted memory [4], power consumption [3, 28], and performance requirements [22, 2]. Current research for *Green IT* (power awareness) strengthens the need for tailor-made, alternative implementations to reduce power costs [9, 15]. For example, a sorting algorithm in an SPL can be implemented to minimize the binary size, maximize the performance, or reduce the power consumptions. However, there is usually no overall best implementation. A stakeholder can decide which implementation of the sorting algorithm fits best to a particular application scenario. Each implementation has a different impact on the non-functional properties of a product which cannot be foreseen. Already the product derivation with a functional selection is a complex task, but how can we provide product derivation support in presence of additional non-functional requirements? The selection of functionality that must be part of a product is not enough any more and the best implementation for an application scenario has to be chosen. Such non-functional requirements can be expressed by defining an objective function to maximize or minimize a set of non-functional properties.

In this paper, we present an approach to optimize products toward non-functional properties. To enable this process, we address two important steps. First, we explain how to measure non-functional properties and how they can be obtained for a feature selection. Second, we present an algorithm to optimize a product configuration according to user-defined non-functional requirements. We evaluate our approach in a case study using the non-functional properties *Maintainability*, *Binary Size*, and *Performance*.

2 Software Product Line Configuration

An SPL is used to derive different related products from a common code base that belong to one domain [10, 19]. Different programs of an SPL differ in features, e.g., one database management system (DBMS) product might have feature *Recovery* and another not (Figure 1). These differences are so called *variation points*. Features of an SPL and relationships between them are described in a feature model with additional information like attributes or annotations [18, 11]. A feature model defines whether a feature is optional or mandatory and is typically visualized with a feature diagram which is a hierarchical representation of all features of an SPL (cf. Figure 1). To derive a product, a stakeholder selects the features from a feature diagram that fulfill her functional requirements. This derivation process is completed with the verification of the correctness of the selection and the generation of the product.

To incorporate non-functional properties, in previous work, we [30] and others [7] extended the common feature model concept. The basis of our product derivation process is a *product line model (PLM)* that distinguishes between domain variation points (features) and implementation variation points (code units) [30]. This gives us the possibility to express variability of the implementation which bridges the gap between functional variation (in terms of features) and non-functional variation (in terms of code units). An implementation variation point is a decision point where a user or an optimizer can decide which implementation of one feature fulfill given requirements. Thus, the user can choose the implementation of a feature that fits best to her non-functional requirements.

Figure 1 depicts a small example of a feature diagram for a PLM representing a DBMS SPL. Feature *Sort* is implemented with two alternative code units, a power saving optimized *JouleSort* [34] and a performance optimized *MergeSort*. When choosing the feature *Data Sorting* the stakeholder has to decide which implementation fits best to the non-functional requirements of the product. In this example, the *JouleSort* algorithm could be selected to minimize the power consumption. In contrast, the *MergeSort* algorithm has a smaller binary size. Considering the binary size of other variation points, it is difficult to decide if a feature selection violates a binary size constraint. For large SPLs this can lead to a time-consuming ”trial and error” configuration process.

3 Measuring Non-functional Properties

Before optimizing configurations of SPL products, we need to measure the non-functional properties. In a case study, we measured non-functional properties for a refactored version of Berkeley DB SPL[1]. Using *feature-oriented programming* [27, 5] the refactoring resulted in 36 features and 400.000 possible products. A simple performance measurement for *one product*, i.e., one complete program was measured, takes about 9 minutes including compiling this product and executing the benchmark. This means a complete measurement including all products would take about 2500 days. Hence, we cannot simply measure each product of this not very large SPL. Moreover, the number of products grows exponentially with the number of features which also brings measurement frameworks like Skoll [26] to its limits.

Instead, we aim at measuring the non-functional properties of features. This is more complex as one would expected, because not every non-functional property can be mapped intuitively to a single feature. For example, some properties emerge only at runtime or depend on the interaction of multiple features. For those properties, still each SPL instance has to be measured to derive the optimal product. In the following, we discuss techniques and a classification for the measurement of non-functional properties.

3.1 Classification of Non-functional Properties

Our analysis has shown that non-functional properties can be categorized into three different classes, *Direct Assigned Properties*, *Inferred Properties*, and *Runtime Properties*. This categorization helps us to select the appropriate measurement technique for a non-functional property.

The first category, **Direct Assigned Properties**, contains properties that are fully or partly represented as features, because the direct assignment by a stakeholder is reasonable. For example, reliability does not emerge during the development or product derivation. It can be directly ascribed to features and allows its configuration according to reliability requirements, e.g., *Recovery* in Figure 1. During measurement, we do not have to care about these kinds of properties because they are assigned manually and can be directly selected.

Non-functional properties of the category **Inferred Properties** are either measured in isolation for each feature or the values for features are inferred from one or few products. This means, we assign values for non-functional properties to features and code units. This allows us to compute emerging non-functional properties for arbitrary product configurations in advance. To do this, we have to ag-

[1]`http://www.oracle.com/database/berkeley-db/db`

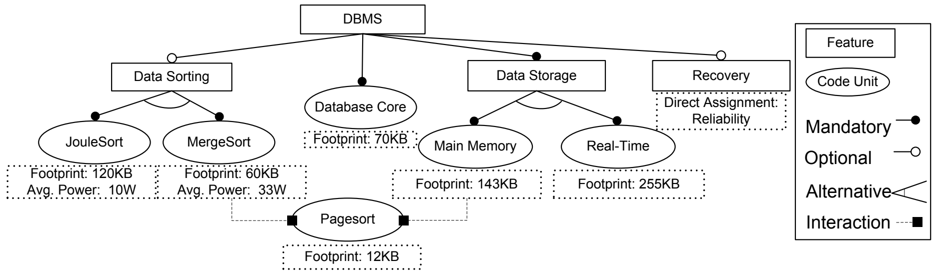


Figure 1. Simple Product Line Diagram.

gregate the values of the properties where the aggregation method depends on the non-functional property. For example, we can measure the binary size per feature and can compute the size of the product by summing the values of each configured feature (see Figure 1). The method used for aggregation has to be defined by a stakeholder and may also depend on the application scenario.

Finally, the category **Runtime Properties** is the most complex class for measurements. These non-functional properties emerge in a running product. Prominent examples are performance, power consumption, and used memory. Feature interactions have a major influence on these properties so that an inference from code units is not possible or has an unacceptable fault rate. Thus, we have to measure these properties in concrete products which leads to a combinatorial problem reasoned by the large number of possible products.

We present our approach for measuring three important non-functional properties, namely *Maintainability*, *Binary Size*, and *Performance*. We chose these properties to present the measurements of the categories *Inferred Properties* and *Runtime Properties*. In addition, these are common properties that are used in practice.

3.2 Measuring Maintainability.

Maintainability describes how much effort is needed to correct, extend, or simply maintain a software system or component [1]. Maintainability is especially important for software development and evolution. To increase maintainability, application engineers structure code units into components or packages. Source code guidelines give suggestions how to write maintainable code [24]. It is a difficult task to quantify the quality of source code. Several metrics were proposed to rate and compare source code fragments. Without loss of generality, we choose the metric cyclomatic complexity [23] for our measurement, but other metrics likes lines of code, comments per line, and combinations of them can be used as well. The resulting values of such metrics can be aggregated. We choose the maximum for cyclomatic complexity to express the difficulty of maintaining the source code, so that the maintainability of the worst module represents the maintainability of the product.

Feature	Cyclomatic Complexity
Cryptography	19
Remove	20
Hash	70
Maximum	70

Table 1. Cyclomatic Complexity of selected Berkeley DB Features.

Source code measurements in SPLs can be applied to code units for each feature separately. Therefore, the maintainability is categorized as an *Inferred Property*. The maximum complexity of the whole feature is the maximum complexity of each method that belongs to this feature. In Table 1, we show a measurement of the Berkeley DB SPL using the tool *Source Monitor*[2] to measure the cyclomatic complexity. A number higher than 25 for the cyclomatic complexity often represents poor written code that might be difficult to understand. That means feature *Hash* appears to be very difficult to maintain. Depending on the configuration, maintainability for the product can significantly change. If the feature *Hash* is chosen, the complexity increases to 70, which means that the code of the resulting product will be difficult to understand. For the computation of the resulting complexity we have to include all scattered code values that belong to a configuration. To compute cyclomatic complexity for an SPL product, we store the measured values for each class fragment that belongs to a certain code unit, separately.

3.3 Measuring Binary Size.

At first sight, it seems that the property binary size has to be measured for each compiled product. However, a mapping from the binary size of a product to features is possible, e.g., by computing the delta size for different prod-

[2]http://www.campwoodsw.com/sourcemonitor.html

ucts. For SPLs implemented by separately compilable code units, e.g., components or Hyper/J [25], can be easily measured for each compiled code unit. Using preprocessor statements or feature-oriented programming, the measurement becomes difficult, because the source code is scattered over modules that cannot be compiled separately. We developed a technique for feature-oriented programming (used in our case studies) to compute the binary size of each feature. First, we include all features into one binary. Second, our tool extracts the binary size for each method from information generated by the Microsoft C++ compiler. Finally, we compute the size of a feature as the sum of all methods that belongs to this feature. Therefore, the binary size can be classified as an *Inferred Property*. We had to disable function inlining to assign the binary size of a method to the correct feature. The impact of function inlining was negligible in our case studies, shown in Section 5, but might be important in other SPLs.

3.4 Measuring Performance.

The last non-functional property, we want to address in this paper, is performance. This non-functional property emerges in a running product, thus, cannot be computed for each feature in isolation in advance (*Runtime Property*). This leads to the combinatorial problem of measuring all possible products of an SPL.

Each application domain or even each application scenario has demands for specific measurements of *Runtime Properties*. For instance, a DBMS can be measured via standard tests like the TPC Benchmark[3] which defines the data to be stored and queries to be executed. We consider a benchmark as a client application that uses the derived product and gives an output that can be evaluated by our tool. The reason for this consideration is the functional difference between derived programs, which may require changes in the benchmark, i.e., explicit function calls to enable the new functionality. If the benchmark is static for all products of the SPL the varying functionality cannot be activated. Only adaptable benchmarks, e.g., also implemented as SPLs, allow for correct measurement of *Runtime Properties* of SPLs.

Interpolation between measured products to reduce the number of performance measurements is not suitable because of unpredictable feature interactions that lead to false interpolated values. For example, consider two sorting algorithms that speed up a program and use large main memory space. Although, both of them increase the performance in isolation and they have no direct interactions, in combination they may degrade the performance because both share the same (too small) main memory. This shows that even features without functional dependencies can have a large influence on a *Runtime Property* if used in combination.

[3]`http://www.tpc.org`

4 Optimization Process

Based on measurements of non-functional properties, we compute an optimized configuration for non-functional requirements. For our optimization process, we adopt the concept of a staged product derivation process [12]. We guide a user stepwise through the configuration process. In each step the number of possible configurations can be decreased. In the first stage, the user selects features based on functional requirements, e.g., the user wants the feature *Transaction*. Existing approaches only focus on this stage. The next stages are part of the optimization of non-functional properties. They are based on the preconfiguration of the first stage, which consists of configured features that have to appear in a product. The optimization task is to find the best implementations for these selected features using a given objective function. In the following process, we use non-functional constraints to restrict the number of products and this objective function to optimize a certain non-functional property. For example, a stakeholder might want to derive a product with best performance while the cyclomatic complexity is less than 25 and the binary size is smaller than 300 KBytes.

To derive an optimized product in reasonable time, we developed several strategies to reduce the complexity of that process. In literature, most approaches are based on *constraint satisfaction problem (CSP)* solvers [7]. We developed a proprietary algorithm that includes several optimization strategies, e.g., using developer knowledge. This is done by comparing developer annotations of alternative implementations and rank them to measure *Runtime Properties* of the supposable best one at first.

In Figure 2, we present an overview of the optimization process. Non-functional properties of the category *Direct Assigned Properties* are directly mapped to features. For the Berkeley DB SPL, we have to consider about 400.000 possible products. In the first configuration stage (feature selection), the number of possible products can be reduced to several thousand, because only these products provide the needed functionality. The number of products depends on the manual selection of functional properties. Members of category *Inferred Properties* are measured before product derivation and are stored in the PLM. In the second stage, products can be excluded because of given non-functional constraints like a restricted binary size. This reduction is possible, because we are able to compute the resulting values for the *Inferred Properties* in advance. In our case studies, only a small number usually lower than one hundred possible products remain. Because this may be still too large for a complete measurement, we sort the configuration (beginning with the best intermediate result of the objective function) for the last stage to measure required *Runtime Properties* of the probably best configuration first. These measurements allow us to compute the final result of the objective function. This step is repeated with a slightly changed configuration (switching between alternative implementations) that may improve the result. Using this process, we find a local optimum. If the number of remaining products allows a complete measurement we find the global optimum. This process is repeated until a user-defined time interval exceeds or the user stops the process.

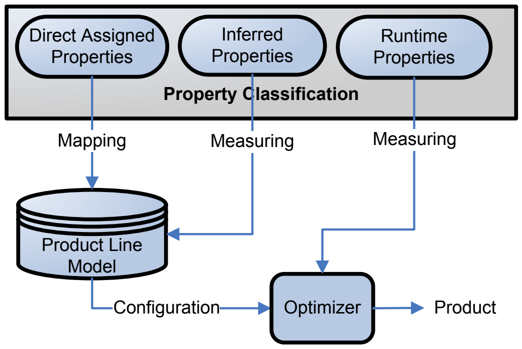


Figure 2. Optimization and Measurement Chain.

5 Evaluation

We evaluate the derivation process to demonstrate the applicability of our measurements and the applicability of the optimization[4]. We use three different SPLs: LinkedList, FAME-DBMS, and Berkeley DB. The first SPL ist the very small *LinkedList* product line which implements a linked list with alternative sorting and traversing algorithms. We use this SPL, because its small size and low number of products allows building and comparing all products with the computed values. The refactored version of *Berkeley DB* [29] and *FAME-DBMS* [29] are chosen, because we want to present evaluation results for SPLs that are used in real environments. The FAME-DBMS SPL prototype implements a flexible, tailor-made DBMS for use in embedded systems like sensors. The number of features, products, and lines of code (LOC) are given in Table 3.

Optimization Process. We measured these three SPLs in two application scenarios to give an impression how much time is needed to measure non-functional properties and to perform the whole derivation process. The first scenario is located in resource constrained environments. In these environments the cost of an embedded system strongly depends on the memory that is required. Even a small reduction of required memory can lead to significant savings in mass production. Usually, a stakeholder defines an objective function that may minimize the production cost of an embedded device. For this scenario, a simplified objective function might be:

$$min\{\mathit{BinarySize}_i\}$$

The binary size is given in Bytes and i represents a product. The second scenario maximizes the maintainability in order to derive an evolvable product while having a binary size and performance constraint:

$$min\{\mathit{Cyclomatic\ Complexity}_i\} \mid \\ \mathit{BinarySize} < 150KB; \mathit{Performance} >= 20T/s.$$

The first stage of reducing the search space for an optimal product is the functional selection of features which is the same for both scenarios. Obviously, this depends on functional requirements of a stakeholder. We randomly selected features in order to simulate such requirements for both scenarios which are shown in Figure 3 (Txn_BDB, Btree_FAME, and Sorting_LL). This random configuration is the result of the first stage. Table 2 depicts the number of possible products after each stage. The first stage has usually the largest impact for the reduction. However, it still results in too many products to measure *Runtime Properties*.

In the second stage, we use the given non-functional constraints. The binary size constraint of the second scenario reduces the number of products as shown in Table 2. This means, that such constraints can have a large impact on the reduction but can also have no impact for too unconstrained non-functional requirements. This depends on the constraint itself and on the products that can be generated from an SPL. The second stage already provides the best product for the first scenario (lowest binary size), thus no runtime measurements are needed. However, in the second scenario the configuration with the lowest cyclomatic complexity for a given performance constraint has to be computed. We can already create a ranking of products based on cyclomatic complexity in the second stage, but we need runtime measures to determine whether they meet the performance constraints. For Berkeley DB, the first product fulfills the constraint and for the other products we only need a small number of measurements (see Table 2).

Time for Measurements. Measuring the properties maintainability and binary size for all features took from 5 to 21 minutes for one SPL. Measuring the cyclomatic complexity took only a few seconds but the manual allocation to

[4]Our evaluations are made on a Pentium 4, 3.00 GHz computer with 2 GB RAM. The installed operation system was Microsoft Windows XP SP2.

	Txn_BDB	Btree_FAME	Sorting_LL
Max Products	400 000	320	480
Stage 1	2000	48	120
Stage 2	10	22	40
Stage 3	1	5	3

Table 2. Reducing the number of products during staged product derivation.

features required a few minutes (see Table 3), but can be automated. To measure the binary size, we compile each SPL one time with all features and code units and automatically allocate the measured binary size of a method to a feature. The compilation requires the largest amount of time. The automatic allocation requires less than one minute for the LinkedList and FAME-DBMS. Performing this task for the larger Berkeley DB SPL takes 3 minutes. These times (see Table 3) are within a reasonable range, considering that the entire SPL is measured, not only a single product.

The whole derivation process, including 5 runtime measurements, requires for the Berkeley DB SPL about 50 minutes. Because of faster compilation the LinkedList takes only 10 minutes for the whole process and FAME-DBMS requires 35 minutes.

Accuracy of Measurements. The accuracy of the measurement, especially when features are measured in isolation and values are aggregated, is very important to retrieve the optimal product that fulfill given constraints. First, cyclomatic complexity is a metric aggregated from measurements of individual methods. Therefore, a aggregation per feature and per product is accurate, as we could also confirm in the generated products.

Second, the performance is measured for each product in isolation which means that this non-functional property is as accurate as the benchmark is but depends on the operating environment and compiler.

Finally, the computation of the binary size of a product is based on values that can change depending on the compiler and the product we want to generate. Therefore, we have to consider the inaccuracy of the binary size computation introduced by compiler optimizations like *function inlining*. To evaluate the accuracy of our approach, we derived some sample products to measure the divergence between the computed binary size and the real binary size including compiler optimizations. Although, the results depend on the implementation of the SPL, we found that the divergence is less than 10 percent. In Figure 3, we present binary size evaluations of two products for each SPL. We measured a minimal version that includes only mandatory features and a large version with more than 10 additional features. For the base (minimal configuration) version of the LinkedList SPL and FAME-DBMS SPL the divergence is less than one percent. Berkeley DB has a higher fault percentage because of a large number of small methods that could be optimized by the compiler. By introducing additional features the divergence of the computed result decreases for Berkeley DB reasoned by functions that cannot be inlined. The error of the LinkedList in the product with additional sorting is reasoned by the small size of methods. These were often inlined in different functions which increases the binary size. This is the only case where the real binary size is larger than the computed binary size and the difference is more than one percent which may lead to measuring *Runtime Properties* of invalid configurations.

Discussion. The main problem of measuring non-functional properties in SPLs and optimizing products for non-functional properties is the large number of possible products. The classification, we have shown in Section 3, allows for measuring some properties without having this problem. However, for *Runtime Properties* there is no solution until now that can provide acceptable measurements regarding the time needed for measurement and accuracy of the results. We still avoid the combinatorial explosion by measuring *Runtime Properties* only for products relevant for a user, i.e., that fulfill given constraints, which is usually a small subset of all possible products. To reduce the number of products, we extended the idea of a staged product derivation [12]. The resulting reduction strongly depends on the non-functional constraints and objective function given by the stakeholder. If only *Runtime Properties* are constrained or have to be optimized we cannot significantly reduce the search space. This means, that the time for measuring *Runtime Properties* can be too large and, thus, derived products may not be optimal. However, we believe that in most cases our approach enables stakeholders to derive products in a reasonable time that fulfill desired non-functional properties.

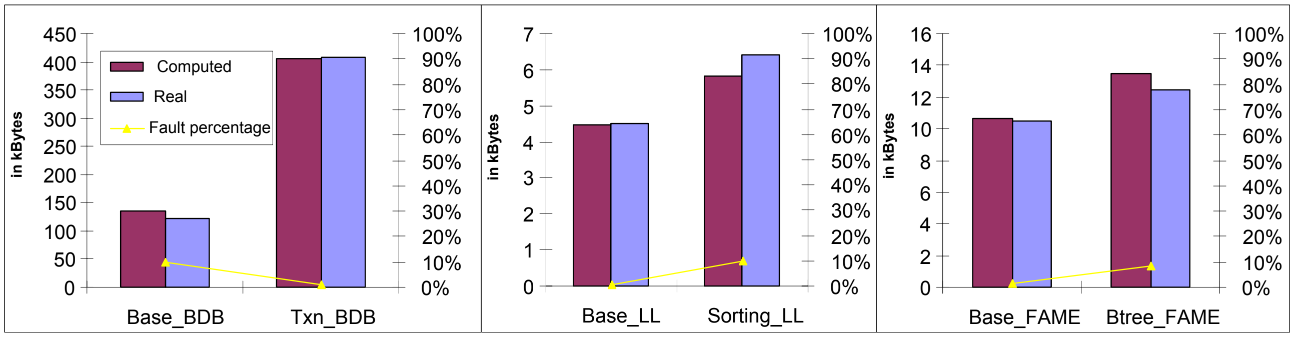
6 Related Work

There are a number of approaches that ease the product derivation process [33, 8, 35]. These tools guide stakeholders through the selection of features with special visualization techniques. However, these approaches concentrate only on functional properties of a product and their dependencies. The measurement of non-functional properties and the configuration of those properties are not addressed.

Some approaches propose techniques on automated reasoning on feature models [7, 6] or SPLs in general [35]. Benavides et al. [7] integrate non-functional properties inside the product derivation process. However, their work

				Time for measuring (in minutes)		
SPL	Features	Products	LOC	Complexity	Binary Size	Avg. Performance (one product)
LinkedList	22	480	886	3	2	<1
FAME-DBMS	21	320	8602	3	6	4
Berkeley DB	36	ca. 400.000	90.198	10	11	9

Table 3. Time spent for measuring the maintainability and the binary size of three SPLs in minutes.



Configuration	Number of Features	Short Description
Base_LL	5	Minimal product of a *LinkedList* SPL
Sorting_LL	12	*Sorting* and *Traversing* features
Base_FAME	15	Minimal in-memory FAME-DBMS version
Btree_FAME	17	*Read* and *Write* support for storage and *Btree* index
Base_BDB	7	Minimal *BerkeleyDB* configuration
Txn_BDB	30	*Transaction*, *Logging*, *Btree*, *Upgrade*, and *Recovery* features

Figure 3. Comparison of computed and real binary size for 6 configurations.

leaves the measurement of the values of these properties as an open problem.

Only a few approaches adapted measurements to SPLs. Either they define the measurements only from a business view to evaluate the development effort [13] or they express only the complexity (in terms of variation points) of the whole product line [21]. An approach close to our work is the measurement of the binary size of an aspect-oriented SPL [17]. Aspects are compiled in distinct files and their size is measured. The binary size of different products can be calculated. However, the approach does not consider other properties or the exponential number of products that occur during the derivation process. The configuration of non-functional properties in SPLs is also addressed by Sincero et al. [31]. This approach tries to overcome the problem of the large number of products by storing the non-functional properties of each derived product in a repository. In addition, automatically generated products further enrich the repository. In contrast, we store non *Runtime Properties* inside a feature model assigned to features. We see this approach complimentary to others and a combination useful. The Skoll project [26] targets on testing and measuring applications with a large configuration space. This project tries to overcome the problem of having a large number of products using a large number of users. We developed a strategy to address the exponential growth of products and, thus, can scale.

7 Summary and Future Work

We presented an approach for measuring non-functional properties in *software product lines* to allow an automatic configuration of code units. We have addressed the problems of measuring diverse non-functional properties by proposing a classification. For each category, we presented examples and showed how these can be measured. We proposed several techniques to reduce the number of possible products by sorting and excluding product candidates in a staged configuring. In an evaluation, we have shown that our approach can significantly reduce the product derivation complexity and the effort to obtain a product including desired non-functional properties. In further work, we will extend our approach to dynamically evaluate runtime properties.

Acknowledgments

This research has been funded by the German Research Foundation (DFG), project number SA 465/32-1[5] and the German Ministry of Education and Research (BMBF), project ViERforES[6].

References

[1] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, 1990.

[2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Trans. on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[3] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proc. Int'l Conf. on Mobile Computing and Networking*, pages 176–189. 2003.

[4] N. Anciaux, L. Bouganim, and P. Pucheral. Memory requirements for query execution in highly constrained devices. In *Proc. Int'l Conf. on Very Large Data Bases*. 2003.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.

[6] D. Benavides et al. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[7] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. *Proc. Int'l Conf. on Advanced Information Systems Engineering*, 2005.

[8] G. Botterweck et al. Towards Supporting Feature Configuration by Interactive Visualization. In *Workshop on Visualisation in Software Product Line Engineering*, 2007.

[9] W. chun Feng, X. Feng, and R. Ce. Green supercomputing comes of age. *IT Professional*, 10(1):17–23, 2008.

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

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

[12] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *Proc. Int'l Software Product Line Conference*, pages 266–283, 2004.

[13] Dave Zubrow and Gary Chastek. Measures for Software Product Lines. Technical Report CMU/SEI-2003-TN-031, Carnegie Mellon University, 2003.

[14] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[15] G. Graefe. Database servers tailored to improve energy efficiency. In *Software Engineering for Tailor-made Data Management*, pages 24–28, 2008.

[16] W. A. Hetrick et al. Incremental return on incremental investment: Engenio's transition to software product line practice. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2006.

[17] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proc. Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 38–45. 2002.

[18] K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SE Institute, Carnegie Mellon University, 1990.

[19] C. W. Krueger. New methods in software product line practice. *Commun. ACM*, 49(12):37–40, 2006.

[20] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. In *Proc. Int'l Software Product Line Conference*, pages 161–160. 2007.

[21] R. Lopez-Herrejon and S. Apel. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering*. 2007.

[22] P. Martin et al. Managing Database Server Performance to meet QoS Requirements in Electronic Commerce Systems. *Int. J. on Digital Libraries*, 2002.

[23] T. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

[24] S. McConnell. *Code Complete, Second Edition*. Microsoft Press, 2004.

[25] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proc. Int'l Conf. on Software Engineering*, pages 734–737. 2000.

[26] A. Porter and C. Yilmaz. Distributed continuous quality assurance: The skoll project. In *Workshop on Remote Analysis and Measurement of Software Systems*, pages 16–19, 2003.

[27] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241, pages 419–443. 1997.

[28] R. Racu et al. Methods for power optimization in distributed embedded systems with real-time requirements. In *Proc. Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, pages 379–388. 2006.

[29] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 1–6. 2008.

[30] N. Siegmund et al. Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties. In *Workshop on Variability Modelling of Software-intensive Systems*, pages 25–31, 2008.

[31] J. Sincero et al. On the Configuration of Non-Functional Properties in Software Product Lines. In *Software Product Line Conference, Doctoral Symposium*, 2007.

[32] M. Steger et al. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Proc. Int'l Software Product Line Conference*, pages 34–50, 2004.

[33] D. Streitferdt, M. Riebisch, and I. Philippow. Details of Formalized Relations in Feature Models Using OCL. *Engineering of Computer-Based Systems*, 00:297–304, 2003.

[34] Suzanne Rivoire and others. JouleSort: a balanced energy-efficiency benchmark. In *ACM SIGMOD Record*, 2007.

[35] J. White et al. Automating Product-Line Variant Selection for Mobile Devices. In *Proc. Int'l Software Product Line Conference*, 2007.

[5] `http://fame-dbms.org`

[6] `http://vierfores.de`