SPL Conqueror: Toward optimization of non-functional properties in software product lines

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel & Gunter Saake

Software Quality Journal

ISSN 0963-9314
Volume 20
Combined 3-4

Software Qual J (2012) 20:487-517
DOI 10.1007/s11219-011-9152-9

Springer

Software Qual J (2012) 20:487–517
DOI 10.1007/s11219-011-9152-9

SPL Conqueror: Toward optimization of non-functional properties in software product lines

Norbert Siegmund · Marko Rosenmüller · Martin Kuhlemann · Christian Kästner · Sven Apel · Gunter Saake

Published online: 28 June 2011

Abstract A software product line (SPL) is a family of related programs of a domain. The programs of an SPL are distinguished in terms of features, which are end-user visible characteristics of programs. Based on a selection of features, stakeholders can derive tailor-made programs that satisfy functional requirements. Besides functional requirements, different application scenarios raise the need for optimizing non-functional properties of a variant. The diversity of application scenarios leads to heterogeneous optimization goals with respect to non-functional properties (e.g., performance vs. footprint vs. energy optimized variants). Hence, an SPL has to satisfy different and sometimes contradicting requirements regarding non-functional properties. Usually, the actually required non-functional properties are not known before product derivation and can vary for each application scenario and customer. Allowing stakeholders to derive optimized variants requires us to measure non-functional properties after the SPL is developed. Unfortunately, the high variability provided by SPLs complicates measurement and optimization of non-functional properties due to a large variant space. With SPL Conqueror, we provide a holistic approach to optimize non-functional properties in SPL engineering. We show how non-functional properties can be qualitatively specified and quantitatively measured in the

N. Siegmund (✉) · M. Rosenmüller · M. Kuhlemann · G. Saake
University of Magdeburg, Magdeburg, Germany
e-mail: nsiegmun@ovgu.de

M. Rosenmüller
e-mail: rosenmue@ovgu.de

M. Kuhlemann
e-mail: mkuhlema@ovgu.de

G. Saake
e-mail: saake@ovgu.de

C. Kästner
Philipps University Marburg, Marburg, Germany
e-mail: kaestner@informatik.uni-marburg.de

S. Apel
University of Passau, Passau, Germany
e-mail: apel@uni-passau.de

context of SPLs. Furthermore, we discuss the variant-derivation process in SPL Conqueror that reduces the effort of computing an optimal variant. We demonstrate the applicability of our approach by means of nine case studies of a broad range of application domains (e.g., database management and operating systems). Moreover, we show that SPL Conqueror is implementation and language independent by using SPLs that are implemented with different mechanisms, such as conditional compilation and feature-oriented programming.

Keywords Software product lines · Non-functional properties · Feature-oriented software development · Measurement and optimization · SPL Conqueror

1 Introduction

A *software product line (SPL)* is a family of related program *variants* that share a common code base (Clements and Northrop 2002). Program variants of an SPL are distinguished in terms of *features*, which are end-user visible characteristics of programs (Czarnecki and Eisenecker 2000). Features usually satisfy functional requirements of stakeholders. Hence, by selecting a set of features, stakeholders derive exactly the variant that fulfills their functional requirements. Common techniques to implement features are conditional compilation (e.g., C preprocessor using #ifdef), components and feature modules (Batory et al. 2004). Features are mapped to these implementation units. According to a feature selection, the corresponding implementation units are used to generate a variant.

Besides functional requirements, stakeholders have requirements regarding *non-functional properties* of a program (Chung et al. 1995). In the literature, the definition of non-functional properties (also referred to as quality attributes) is not consistent (Robertson and Robertson 1999; Glinz 2007; Chung and do Prado Leite 2009). We use the definition of Robertson and Robertson (1999), who define a non-functional property as: *"A property, or quality, that the product must have, such as an appearance, or a speed or accuracy property."* We focus on common non-functional properties such as performance, reliability, footprint and so forth. When developing single programs, non-functional requirements are identified and documented *before* product development (Chung et al. 1999). During development, tools such as the non-functional requirements framework (Chung et al. 1999), i* framework (Yu 1997) and KAOS (van Lamsweerde 2001) help developers with design decisions that affect non-functional properties of the final program variant. Conflicting requirements have to be resolved during development (van Lamsweerde et al. 1998). SPLs change this picture.

In contrast to conventional software development, an SPL usually covers a broad spectrum of application scenarios in a certain domain. A vendor develops an SPL for an entire domain in which stakeholders can have very different non-functional requirements. Depending on the concrete application scenario, it is even possible that customers have conflicting or contradicting non-functional requirements. Hence, an SPL has to satisfy very different non-functional requirements. As running example, consider an SPL of *database management systems (DBMS)*. Stakeholders of such an SPL (e.g., users who derive a variant of an SPL) have completely different non-functional requirements when they use a particular variant in different application scenarios, such as mobile devices, parallel computers, or desktop computers. For example, the footprint of a DBMS variant has to be minimized for an embedded system, a variant for real-time systems must provide a

deterministic response time, and a DBMS variant for a mobile device requires minimized energy consumption.

Contrary requirements result in alternative features, which in turn result in the ability to provide different variants that satisfy even contradicting requirements. For instance, a DBMS SPL may provide alternative buffer manager features, e.g., a feature minimizes working memory consumption and another feature optimizes performance. Hence, many non-functional requirements are defined when deriving a concrete product during application engineering, i.e., *after* SPL implementation (a.k.a. domain engineering). Non-functional as well as functional requirements can often only be defined *per application* or *per customer*, that is, at product derivation time. As described before, highly differing and even contradicting non-functional requirements of different concrete application scenarios make it necessary to postpone the definition of objective functions (or quality goals) to the product derivation phase. Hence, often SPL vendors do not know the concrete non-functional requirements before variant derivation (i.e., after development) and can only prepare an SPL for anticipated possible requirements.

As a result of the high variability of an SPL, it is usually not clear which feature selection leads to which non-functional properties. Since variants are generated by selecting desired features, it is difficult to predict which selection of features or which alternative feature implementations result in a program variant with, for example, a footprint lower than 200 KB and a response time of less than one second. Again, an SPL cannot be tuned to these requirements as the requirements are usually not known beforehand and vary (in certain bounds) depending on the application scenario, environment and customer. Therefore, SPL developers implement a spectrum of non-functional properties with a large degree of freedom in the implementation. During product derivation, actually important properties are determined by individual application engineers. Hence, application engineers often face the questions: Is there a variant that meets my functional requirements and also satisfies my non-functional requirements? What is the best trade-off between different properties?

Answering these questions is far from trivial. An SPL usually has many variants that satisfy the same functional requirements. To give a correct answer, an SPL's vendor would have to measure the properties for all of these candidate variants. This can lead to a costly and time-consuming trial and error process, because even small SPLs with only few features can have millions of possible variants. With an increasing number of features, vendors face an exponential explosion of the variant space. Generating, compiling and executing each relevant program is infeasible even for medium-sized SPLs (Krueger 2006; Siegmund et al. 2008b). Even worse, some non-functional properties cannot be measured at all. They have to be described qualitatively on an ordinal scale. These kinds of properties must be considered for variant derivation, too.

Besides measuring and determining non-functional properties of a variant, a customer often wants to derive a variant that is optimized with respect to a specific non-functional property (e.g., performance or footprint). This means also that it is not sufficient to find a feature selection that meets the requirements, but to calculate the *optimal* feature selection for a given non-functional property.

We present a holistic approach, called *SPL Conqueror*, that integrates all aspects of the variant-derivation process with respect to non-functional properties. We show the big picture of optimizing non-functional properties in the area of SPLs. SPL Conqueror supports the optimization of qualitative and quantitative properties. In previous work, we focused only on certain aspects of the optimization process. In this paper, we combine developed solutions to automate the whole optimization process. We subsume and extend

our previous work and evaluations (Siegmund et al. 2008a, b) and make the following novel contributions:

1. We extend our integrated product-line model (Siegmund et al. 2008a) to assign non-functional properties to features and implementation artifacts and to model feature interactions explicitly. SPL Conqueror uses this model to compute an optimized variant based on a feature's properties.
2. We classify non-functional properties into three classes (*qualitative, feature-wise quantifiable* and *variant-wise quantifiable properties*). We use these classes to select suitable measurement and configuration techniques.
3. With SPL Conqueror, we provide a holistic approach in which user-defined metrics are used to measure different non-functional properties. During measurement, we address the problems of the variable code base of SPLs. Furthermore, we *automate* the measurement process and enrich an integrated product-line model with the measurement results.
4. We support derivation of variants that are optimized with respect to non-functional properties by (1) highlighting features that improve a certain non-functional property, (2) predicting the value of a non-functional property for a variant based on approximations of a feature's properties and (3) automatically measuring promising variants using our automated measurement framework.

2 Problem statement

In this section, we describe the challenges of measuring and optimizing non-functional properties in SPLs. This is the basis for understanding the rationales behind our classification and measurement approach.

2.1 Software product line scenario

In SPL development, we differentiate between *domain engineering* and *application engineering* (Czarnecki and Eisenecker 2000; Pohl et al. 2005) as illustrated in Fig. 1. A domain engineer analyzes the functional and non-functional requirements that are

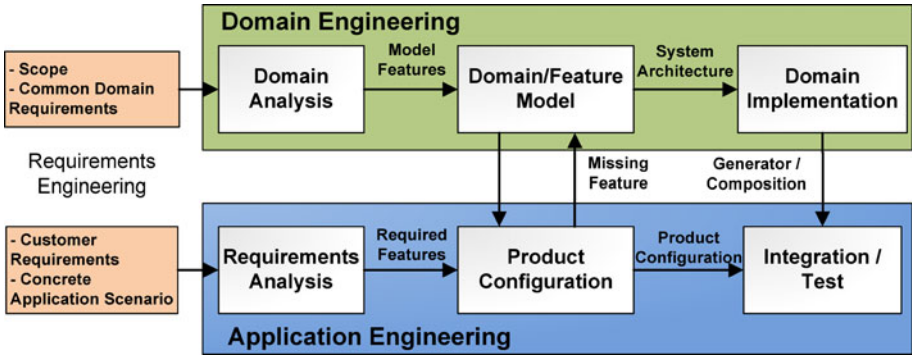

Fig. 1 Domain and application engineering phases in SPL development including requirements specification (Czarnecki and Eisenecker 2000)

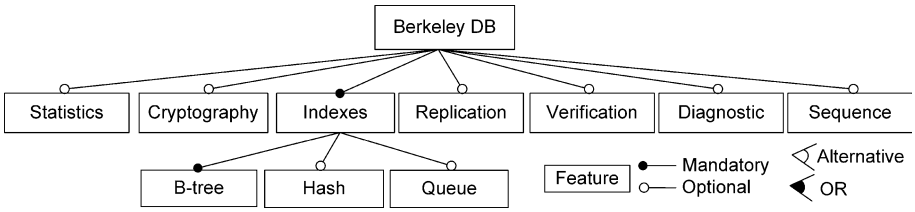

Fig. 2 Feature model of Berkeley DB (C version)

important for an entire domain (i.e., not necessarily for a single application scenario). 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 (i.e., performance vs. reliability), they both are useful for specific scenarios (e.g., an in-memory variant for a web browser and a persistent variant for an e-mail client). That is, developers implement alternative features to satisfy different and even incompatible goals. After domain analysis, developers and domain engineers design and implement reference architecture for the SPL. Hence, typically almost the whole implementation work is done in the domain engineering phase (Clements and Northrop 2002).

For each product, application engineering starts with the requirements analysis of a concrete application scenario. After this second requirement engineering phase, an application engineer (e.g., the customer) selects features to satisfy her requirements. If requirements cannot be satisfied (e.g., because functionality is missing or non-functional requirements cannot be fulfilled), 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 (e.g., a feature requires the presence of another feature).[1]

In Fig. 2, we visualize the feature model of the Berkeley DB SPL. We use Berkeley DB as a running example. Berkeley DB[2] is a customizable DBMS with over 200 million deployments (Oracle 2006). It has optional features (e.g., Hash, Queue, Cryptography) to be able to tailor a program variant to a customer's requirements. One can generate 256 different variants for the Windows platform. Features are represented by boxes and connections between them express domain constraints. For example, a feature connected by an empty bullet is optional (e.g., feature Hash), and a feature connected with a filled bullet is mandatory (e.g., feature B-Tree). There are also grouping relationships in a feature model. For example, a set of features can be alternative (XOR), which enforces user to select exactly one feature of the alternative group. Furthermore, we can define an OR group that allows user to select between at minimum one and an arbitrary number of features.

[1] Please note, a feature model looks similar to a goal model often used for requirements engineering (van Lamsweerde 2001). However, the concepts cannot be compared. A feature model describes the variability of an entire SPL, i.e., all products.

[2] Available at: http://www.oracle.com/technetwork/database/berkeleydb/.

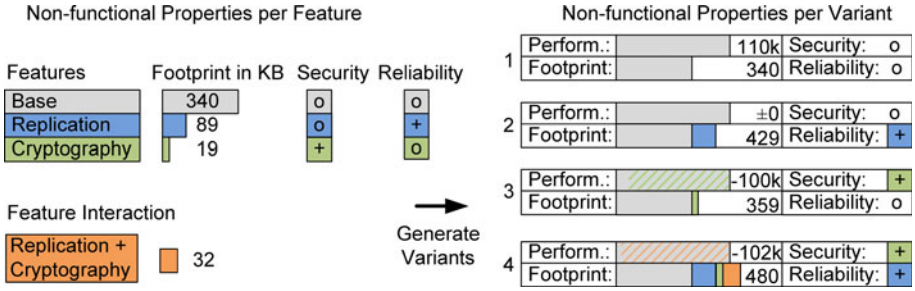

Fig. 3 Relationship between non-functional properties and feature interactions. On the *left side*, we depict the non-functional properties per feature (e.g., feature Replicate has a footprint of 89 KB). On the *right side*, we show the measured properties of all variants that can be generated with the three features. For footprint, we show the composition of the features' footprints of a variant. The feature interaction introduces additional footprint and changes the performance of variant 4. A plus symbol describes qualitatively that a feature improves a certain property. Performance is given in transactions per second (T/s) and footprint in KB

2.2 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 (Stevens 1946). We measure non-functional properties that can be described with a metric scale (i.e., interval and ratio) for which a stakeholder (i.e., a vendor, developer, domain expert, or user) can define a suitable metric. For example, we can define footprint (using the measurement of the binary size) 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 Sect. 3.1.

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. In Fig. 3, 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 89 KB. 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 non-functional properties. On the right side of Fig. 3, we show program variants with different feature combinations. The variant that includes both features Replication and Cryptography have 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 448 KB[3] rather than

[3] The sum of the footprint of features Base, Replication and Cryptography is 448 KB.

480 KB. The observed difference is caused by feature interactions of both features at the source code level.

Another example is *SQLite*.[4] 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 2^{88} different variants. Measuring all variants would take longer than the time the universe exists.[5] Obviously, a customer cannot find the optimal variant with a brute force approach.

2.3 Optimizing non-functional properties

Optimization means to find the best variant (feature selection) according to specific non-functional properties. To optimize a variant with respect to non-functional properties, we can search for an optimal feature selection during *application engineering*. For example, we select those features that have the most positive influence on a property. For example, we would select the B-tree search index in Berkeley DB (cf. Fig. 2) to optimize performance. However, there are usually trade-offs between non-functional properties. Selecting feature B-tree increases the binary size, which might be not acceptable for some application scenarios. Typically, an SPL vendor has to cooperate with customers to define an objective function over a set of non-functional properties. An objective function expresses how to rate the diverse non-functional properties to achieve the desired goals (van Lamsweerde et al. 1998; van Lamsweerde 2001; Marler and Arora 2004).

Another problem is the computational complexity of finding an optimal variant (Floch et al. 2006). White et al (2009) found that this problem is NP-hard. Special algorithms are needed to *approximate* a good solution. Although there are already some solutions available (e.g., using filtered Cartesian flattening (White et al. 2009) or constraint satisfaction problem solvers (Benavides et al. 2005)), they work only for a limited class of properties (which we later describe as *feature-wise quantifiable*). Other properties, such as performance and energy consumption, that can only be measured per variant are not addressed and neither are qualitative properties (e.g., user-friendliness). Thus, we require a combined approach of computing optimized variants on a *per feature basis* and measuring non-functional properties on a *per variant basis*. We show how our approach addresses this issue in Sect. 6.

3 Representing non-functional properties in software product lines

We aim at optimizing non-functional properties in the product-derivation phase. Defined by an SPL vendor or customer, the specification of desired properties must either contain a

[4] Available at: http://sqlite.org.

[5] In fact, a single measurement takes approximately 5 min. Measuring all 2^{88} variants would take ca. 2.9×10^{21} years.

qualitative statement regarding the range of values of a property or a metric that we can use to measure a property. Hence, we need the information whether a property can be described with an ordinal or a metric scale. To this end, we categorize non-functional properties based on measurement theory (Stevens 1946) to use the proper measurement and derivation technique for a given non-functional property.

3.1 Classification of non-functional properties

There is a number of non-functional properties including their classification described in the literature, for instance, McCall's quality model (Mccall et al. 1977), Boehm's quality model (Boehm et al. 1978) and the ISO 9126 quality model (International Organization for Standardization (ISO) 2001). These models have a certain purpose. For example, McCall's quality model bridges the gap between a customer's quality perspective and a developers view on quality attributes. Hence, McCall describes factors based on an external view of a software and quality criteria that describe the internal view of a software. A developer can use this model to derive suitable metrics (e.g., error tolerance and accuracy) to improve a quality factor (reliability). Boehm's quality model is a hierarchical model to refine and further specify characteristics from which a property is composed (Boehm et al. 1978). For example, maintainability is refined to understandability which in turn is refined to conciseness. Hence, he qualitatively defines software quality with a given set of metrics.

In contrast to the mentioned models, our purpose is to classify non-functional properties such that we can choose proper optimization techniques based on this information. Some non-functional properties can be described only qualitatively, whereas other properties can be represented with metric-based values, so we cannot use the same optimization technique for all properties. For example, we cannot *compute* which feature selection results in a variant with the best user-friendliness, because we usually have no metric to obtain quantifiable measures. But, we can compute the variant with the smallest footprint or highest performance. Hence, we classify non-functional properties with respect to our ability to measure them and which operations are valid for the measures.[6]

We classify non-functional properties into three different classes: *qualitative properties, feature-wise quantifiable properties* and *variant-wise quantifiable properties*. It is important to note that the categorization of a specific non-functional property depends on the SPL and the application scenario and is not general. This means that the same property can be in different classes for different SPLs or domains. Reasons for different classifications are, for example, different view points and interpretations of stakeholders for the same property. Also, the domain of an SPL may change the category of a property. For instance, in a web-service SPL, security may be measured via an intrusion-detection system resulting in quantifiable measures. In another scenario, security can only be qualitatively specified (e.g., with weak, medium and strong secure), like it is done in Window 7.

3.1.1 Qualitative properties

There are non-functional properties that can only be described qualitatively using an ordinal scale (i.e., there is no metric from which we can retrieve quantifiable measures). For example, we can define that feature Verification in Berkeley DB improves the

[6] Measurement theory defines which operations are valid for which scale of measurement. For example, we can only use median and percentile operations for an ordinal scale, because we only have a totally ordered set of measures.

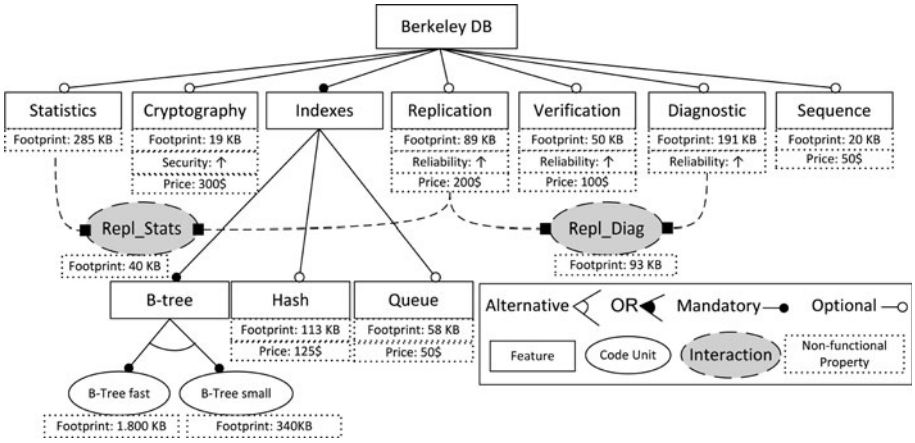

Fig. 4 Product-line model of Berkeley DB with assigned properties. Footprint represents the actually measured binary size per feature. The *up-arrow* visualizes an improvement for a qualitative property

reliability of a DBMS, because it verifies the consistency of indexes (cf. Fig. 4). We can assign such a qualitative statement to features (i.e., "feature Verification improves reliability"). Since ranking is a valid operation for values on an ordinal scale, a domain expert can rate features according to their influence on a non-functional property. For example, we can rate feature Verification higher than feature Diagnostic for property reliability, because in the DBMS, domain consistency of search indexes has a crucial impact on reliability whereas Diagnostic functions may only identify some possible weak points for reliability.

Qualitative properties usually require domain knowledge. Hence, SPL vendors should define important properties that can be used by customers as guidelines to support feature decisions during product derivation. For example, we can assign them certain values (e.g., Verification = 2 and Diagnostic = 1). Again, a stakeholder must keep in mind that only certain types of calculations (median, percentile) are suitable over ordinal numbers (Stevens 1946). To sum up, we use qualitative properties (a) to show them as configuration possibilities to the user (e.g., hint which features qualitatively improve a certain property), (b) to automatically select features with positive influence, and (c) to avoid the selection of features with negative influence during the computation of an optimal variant.

Common representatives of this class are: *reliability, security, trustability, availability, usability, integrity and completeness.*

3.1.2 Feature-wise quantifiable properties

This category contains properties that can be measured on a metric scale. An important requirement for feature-wise quantifiable properties is that we can either measure a single feature directly or infer the results of the measurement of a variant to single features with an user-defined metric (i.e., either customers or SPL vendors provide suitable metrics). Hence, we can compute to which extent a feature influences a non-functional property. Examples of this class are footprint of a feature (which can be measured per implementation unit (Siegmund et al. 2008b)) and maintainability (which can be measured to some degree with code metrics such as lines of code and cyclomatic complexity (McCabe

1976)). A feature-wise measurement allows us to annotate each feature and implementation unit of an SPL with a specific value and to compute a value for a feature selection. To compute a value for a concrete feature selection, a stakeholder defines an aggregation function, such as addition or maximum. The aggregation function is used to aggregate the values for each selected feature. For example, we defined for Berkeley DB *maximum* as aggregation function for cyclomatic complexity and *addition* for footprint. This way, we are able to compute the properties of a variant in advance only based on a configuration.

Common representatives of this class are: *footprint, maintainability, accuracy/resolution of data, price of a feature, adaptability, interoperability and modularity*.[7]

3.1.3 Variant-wise quantifiable properties

Some properties have either no meaning for single features or we are not able to quantify the influence of individual features on the non-functional properties of a concrete variant. Usually, such properties emerge when a variant is executed. They require the highest measurement effort, because we have to generate each variant from which we want to know properties. This usually requires to execute and to measure a variant, e.g., by running benchmarks. For example, to measure performance in Berkeley DB, we use Oracle's standard benchmark, which defines certain types of queries. Considering the large number of possible variants, variant-wise properties should be measured only for a predefined set of selected features. This set may be the result of previous optimization and configuration steps based on the properties of the previous categories. Similar to the previous class, variant properties can be described with a metric scale.

Common representatives of this class are: *performance, response time, resource behavior (e.g., energy and memory consumption) and bandwidth*.

3.2 Product-line model to reason about feature selections

As we explained before, a feature model describes the variability of an SPL and ensures that only meaningful variants can be derived (Kang et al. 1990; Czarnecki and Eisenecker 2000). We extended the common feature-model approach to include also non-functional properties of features and implementation units (Siegmund et al. 2008a). We call our extension a *product-line model*. In Fig. 4, we show the product-line model of Berkeley DB. In addition to the feature model of Fig. 2, we model implementation units. For instance, B-Tree fast and B-Tree small are alternative (mutually exclusive) implementations of feature B-tree.

The product-line model supports the assignment of *qualitative properties* (with ordinal values) and *feature-wise quantifiable properties* (with actually measured metric values). In Fig. 4, we show the footprint of each feature that we measured for Berkeley DB's features (described in Sect. 5) We assigned also a price for features, for illustration. Furthermore, we defined two qualitative properties security and reliability and also highlighted that a certain feature has a positive influence on this property. For example, feature Verification has a positive effect on reliability for a DBMS.

We distinguish between alternative features and alternative implementations. While alternative features define different functionality, alternative implementation units implement the same functionality in different ways. For instance, an user can decide either to derive a performance-optimized Berkeley DB variant (by selecting the implementation unit

[7] Maintainability can be derived from source code metrics to some degree.

B-Tree fast) or a footprint-optimized (binary size) variant (by selecting the implementation unit B-Tree small). Hence, alternative implementations represent variability at the level of non-functional properties. Often, alternative implementations are extensions for new customers who have new requirements that cannot be satisfied with the currently available SPL implementation. If an user is not interested in a non-functional property, then often a standard decision is made.

As an important extension to our product-line model (Siegmund et al. 2008a), we introduce the concept of *feature interactions* in our product-line model. Feature interactions change non-functional properties of a feature depending on the presence of a certain feature combination. We explicitly model feature interactions to consider them for predicting a variant's non-functional properties. For example, interactions occur when multiple features share a common code unit or when a certain feature combination requires additional code (e.g., using nested #ifdefs). Additionally, feature interactions can cause deadlocks and bus overloads. In Berkeley DB, there is an exhaustive use of nesting a feature's code in another feature's code (e.g., to implement statistics for the hash search index; cf. Fig. 2) resulting in different binary sizes depending on a certain feature combination. In Berkeley DB, we identified a feature interaction between features Replication and Statistics. We measured the influence of this interaction on footprint: A product with both features in combination has an increased binary size of 80 KB in addition to sum of the feature's sizes. Such feature interactions occur for many non-functional properties.

4 SPL Conqueror: a holistic approach for the optimization of non-functional properties

With SPL Conqueror, we propose a holistic approach to integrate measurement and optimization of non-functional properties in the product-derivation process. With holistic we mean that we support the whole product derivation process starting from the definition of desired non-functional properties, over the measurement of properties, to the concrete feature selection and optimization by means of an objective function. We support the different kinds of non-functional properties described in Sect. 3.1. A stakeholder (i.e., an SPL vendor or domain expert) can assign properties to features to describe the influence of a feature on a specific property. In addition, a stakeholder can specify measurements and metrics in SPL Conqueror to measure either a single feature (e.g., the source code complexity) or a whole variant. Once the measurement procedure is defined, the process of selecting features and generating and measuring features is automatically performed.

The results of measurements are stored in the SPL's product-line model, which we described in Sect. 3.2. We use this model including all assignments and measurements during the product derivation phase to provide multiple optimization possibilities. Customers can define non-functional constraints (e.g., a footprint limit of 200 KB) as well as objective functions for quantifiable properties (e.g., maximize performance). If the objective function contains a property that can only be quantified on a per variant basis, SPL Conqueror automatically generates and measures variants to identify the optimal variant. In Fig. 5, we provide an overview of the process of SPL Conqueror including the following tasks (cf. Fig. 5):

- (a) Assign quantifiable properties to features (by domain expert)
- (b) Measure non-functional properties per feature (by domain expert and vendor)

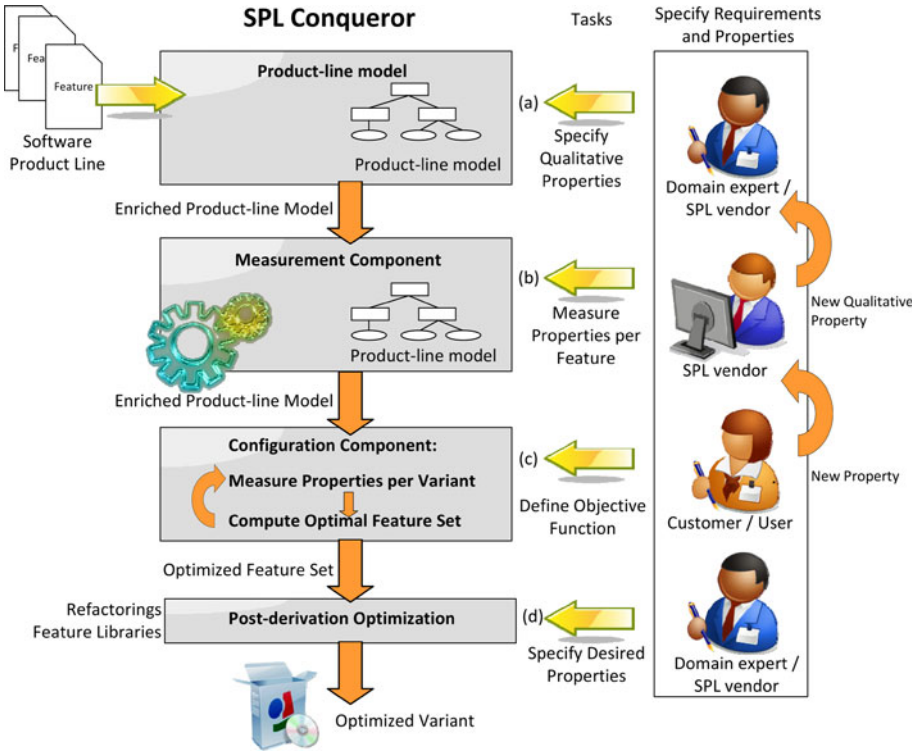

Fig. 5 Process of SPL Conqueror including the different tasks of measurement, configuration and optimization

- (c) Define functional and non-functional requirements in application engineering per variant (by customer)
- (d) Optionally apply additional post-derivation optimizations to a generated variant (by domain expert and SPL vendor)

In the Sects. 3–6, we describe each task in detail. Analogously to the classification of non-functional properties (which we described in 3), we have different tasks to specify and measure non-functional properties. For qualitative properties, a domain expert assigns non-functional properties to features (in Fig. 5a). Ordinal values are stored in the product-line model and are used in the variant-derivation process.

The next step is to measure quantifiable properties per feature (in Fig. 5b), which we describe in Sect. 5 For this task, an SPL vendor or domain expert defines proper measurement procedures (e.g., a source code metric or a tool which measures a property). These measurement procedures are plugged into SPL Conqueror to automatically measure individual features and to store the values in the product-line model. We describe the measurement of properties of all categories including its evaluation based on multiple SPLs in Sect. 5.

To derive a variant (in Fig. 5c), a customer defines functional and non-functional requirements. That is, she selects features satisfying functional requirements and defines constraints for non-functional properties as well as optimization goals (in terms of

objective functions). For example, if a customer wants to optimize the footprint of Berkeley DB, she would define an objective function, such as *min(Footprint)*. During the derivation process, SPL Conqueror provides for each property class a specific configuration and optimization technique. We describe each technique including an algorithm in Sect. 6 in detail.

Once SPL Conqueror has found an optimal feature set (in Fig. 5d), an SPL vendor can apply further optimizations to this variant. In the past, we developed two techniques that scale to large SPLs and a high number of properties considered for optimization (Siegmund et al. 2010a, b). We can purposefully use refactoring to alter the structure of a generated variant in such a way that a certain non-functional property is improved. Our second technique uses libraries of features that realize specific optimizations for different non-functional properties. By linking such additional features in a variant, we can optimize a non-functional property.

We structure the remaining article according to the tasks of SPL Conqueror. We first describe for each class of non-functional properties our measurement techniques and explain how we realized measurements in our case studies and which experience we gained. We continue with a demonstration of the variant-derivation process.

5 Measuring non-functional properties

The measurement of non-functional properties is a challenging task, because often one cannot measure features in isolation (i.e., without the presence of and interaction with other features), and we have to guarantee that the measured feature is actually used in the benchmarked variant. We illustrate our approach for the measurement of *reliability, complexity, footprint* and *performance*. We selected these non-functional properties, because they are commonly relevant during variant derivation, and they are representatives of quantifiable and qualitative properties. In Fig. 6, we show the dialog of SPL Conqueror with which users can define measurement procedures and metrics for a specific property. In Fig. 6a, an user defines a measurement procedure using the following parameters: the program that performs the measurement, required input values, and an access method to extract the measurement values from the program output. Furthermore, aggregation instructions can be inserted (Fig. 6b) to define how the values obtained from individual features must be aggregated to obtain a value for an entire variant. SPL Conqueror can export and import such definitions as XML files. Thus, metrics and measurement specifications can be reused in different contexts and easily exchanged.

In previous work, we evaluated the measurement process using SPL Conqueror with nine existing SPLs that have very different characteristics to cover a broad spectrum of scenarios (Siegmund et al. 2011).[8] In this paper, we present an approach to compute non-functional properties of features based on a small number of generated and measured variants. In contrast to previous work, we only give here examples for how measurements can be achieved with SPL Conqueror rather than presenting concrete algorithms to compute a feature's properties. We give an overview of the sample SPLs, in Table 1. We selected case studies of varying sizes (2500 to 13 million lines of code, 5 to 100 features) and implemented with different languages (C, C++, and Java) and different variability mechanisms (conditional compilation and feature-oriented programming), from different

[8] We provide the raw material of our measurements and evaluations on our website: http://fosd.de/SPLConqueror.

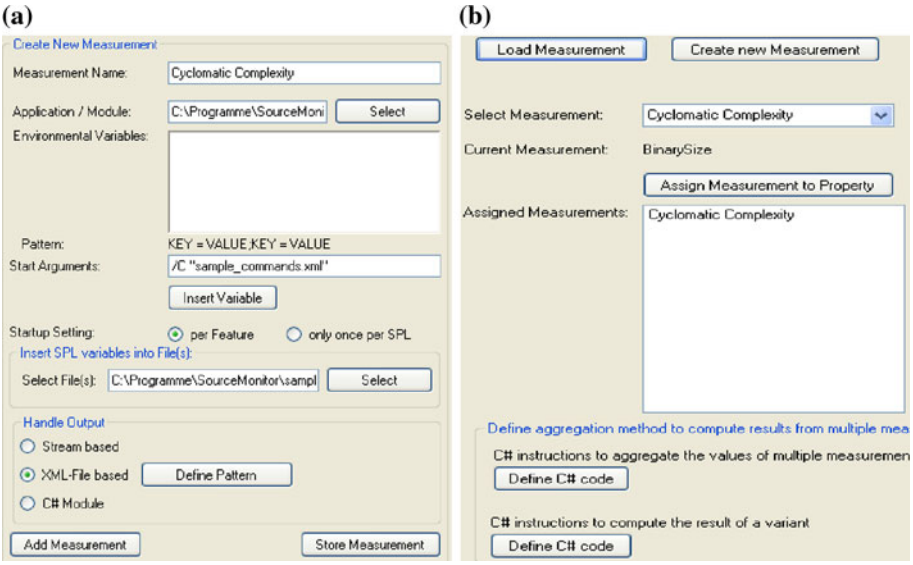

Fig. 6 Creating a measurement for a feature-wise quantifiable property in SPL Conqueror. In **a**, users can define a measurement procedure. In **b**, users can specify their aggregations functions for aggregating non-functional properties of selected features

Table 1 Overview of the SPLs used in our evaluation. OS: Operating system; Acad.: Academic; Ind.: Industrial

Product line	Domain	Origin	Lang.	Features	Variants	LOC
Linked list	Structures	Acad.	Java	18	492	2,595
Prevayler	Database	Ind.	Java	5	24	4,030
ZipMe	Compr. lib	Acad.	Java	8	104	4,874
PKJab	Messenger	Acad.	Java	11	72	5,016
SensorNet	Simulation	Acad.	C++	26	3240	7,303
Violet	UML editor	Acad.	Java	100	ca. 10^{20}	19,379
Berkeley DB	Database	Ind.	C	8	256	209,682
SQLite	Database	Ind.	C	85	ca. 10^{23}	305,191
Linux kernel[a]	OS	Ind.	C	25	ca. 33×10^6	13,005,842

[a] We use only a subset of 25 features of the Linux kernel selected by a domain expert

domains (e.g., operating systems, database engines and end-user applications) and from different developers (both, academic and industrial).

5.1 Reliability (Qualitative property)

To specify a qualitative property, a domain expert inserts only the name of the property in SPL Conqueror and selects the features and implementation units that improve or degrade the property. Additionally, the domain expert can rank the features according to their influence on the property.

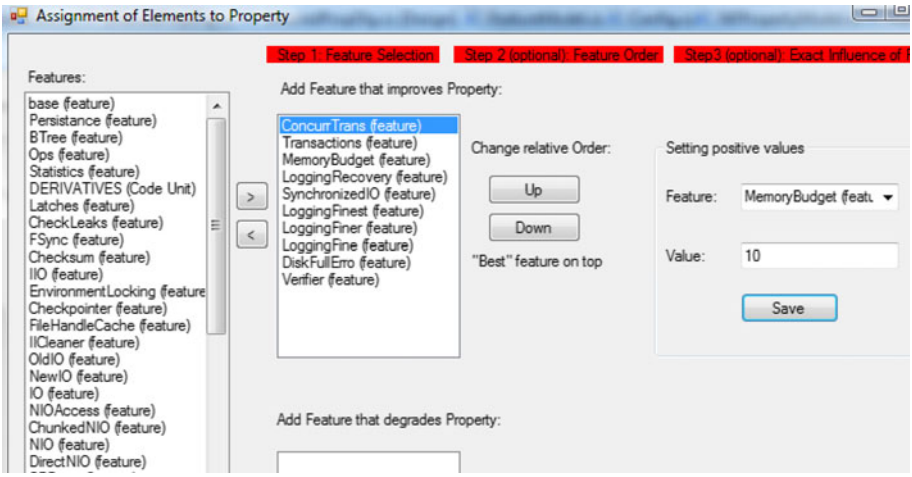

Fig. 7 Definition and assignment of the qualitative property reliability in SPL Conqueror

Giving an explicit ranking of features and implementation units may not be sufficient for a later (automatic) optimization. For example, Berkeley DB has multiple features that improve reliability, but the effect can heavily differ (as we described earlier). Hence, we provide the option to define a value for each feature. These values estimate the impact of a feature on the property. Similar to a feature model, in which features are an agreement about expected functionality between stakeholders of a domain, we consider non-functional properties as agreements between stakeholders about the meaning of a property (Czarnecki et al. 2006).

In Fig. 7, we show the assignment of the non-functional property reliability to a number of features. For example, we define that feature concur_transaction, transactions, and loggingrecovery have a positive influence on reliability. Furthermore, we define values for these features to express their influence (e.g., feature memorybudget and loggingrecovery have both the same value). This means, that both features have the same positive effect. If other non-functional requirements are relevant in the optimization process (e.g., a footprint constraint), we are able to decide whether to select memorybudget or loggingrecovery, depending on their measured footprint.

5.2 Measuring the complexity of a feature's source code (feature-wise quantifiable property)

Knowing the complexity of a feature's source code is important if an SPL vendor sells the source code of a variant including the responsibility to maintain the variant. A typical application scenario in which a customer is interested in the source code of a variant is components (e.g., graphical components). Often, a component must be further customized and so must the source code of a component adapted, e.g., in white box frameworks and libraries. To make the adaptation process efficient, a customer is interested in buying a component that is easy to understand, maintain and customize. Hence, we have to provide means that allow customers to derive easily maintainable variants.

There are several metrics that measure the complexity or maintainability of source code. We use McCabe's cyclomatic complexity (McCabe 1976) as an exemplary metric, but

Table 2 Lines of code (LOC), maximum and average cyclomatic complexity of Berkeley DB and ZipMe SPLs

Feature	Complexity			Feature	Complexity		
	LOC	Max.	Avg.		LOC	Max.	Avg.
B-tree	18,223	39	3.8	Base	2,837	25	2.8
Hash	11,562	14	4.1	Adaptation	7	1	1
Queue	7,394	23	3.3	Checksum	169	1	1
Sequence	913	12	2.8	ArchiveCheck	17	4	1.4
Verify	8,924	25	4.1	Compress	679	30	4
Statistic	9,576	63	3.7	CRC	34	5	1.8
Cryptography	1,058	1	1	Extract	293	17	3.4
Replication	9,112	8	2.2	GZIP	190	26	3.6
Diagnostic	21,342	15	2				

other metrics could be used as well. For measurement, we used the tool Source Monitor[9] (cf. Fig. 6). For example, for SPLs implemented with feature-oriented programming[10] the source code of each feature is physically separated in different folders. When starting the measurement, SPL Conqueror executes Source Monitor for each feature (giving the folder of the feature's source code as input), extracts the XML output with a previously defined XPath statement and stores the result in the product-line model. Either a standard aggregation function (we use the maximum) or a user-defined aggregation function is used to determine the complexity of a feature from the complexity values of the classes of the feature. As a design decision, we define the complexity of a feature as the maximum complexity of each method that belongs to this feature.[11]

In Table 2, we show the results of two SPLs. We omit measurements of other SPLs, since the results are straightforward.[12] A number higher than 25 for the cyclomatic complexity is considered to be poorly written code that is difficult to understand (McCabe 1976). For example, feature Compress of ZipMe (implementing a hash-based search index) appears to be very difficult to maintain according to this metric with a measured value of 30. Depending on the configuration, maintainability for the variant (according to such metrics) can significantly change.

5.3 Measuring footprint (feature-wise quantifiable property)

Measuring the footprint of an application strongly depends on the used implementation technique. There are many ways to measure the footprint of a feature. For example, we developed in previous work (Siegmund et al. 2008b) two methods to measure the footprint of an SPL implemented with feature-oriented programming (Batory et al. 2004). When using an implementation technique that supports separately compilable code units, e.g., components or feature modules (Batory et al. 2004), we can easily measure these units and

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

[10] The SPLs are: LinkedList, ZipMe, PKJab, SensorNetwork and Violet.

[11] Please note, that this is only an example. Such a design decision should be made by domain experts and SPL vendors.

[12] The complete measurement can be found at our website: http://fosd.de/SPLConqueror.

Table 3 Approximated footprint (binary size) of selected features of Berkeley DB, Linux kernel and ZipMe

Berkeley DB		Linux kernel		ZipMe	
Feature	Footprint (KB)	Feature	Footprint (KB)	Feature	Footprint (KB)
B-tree	1,800	SMP	709	Base	79
Hash	113	INotify_User	11	CRC	1.6
Queue	58	Firmware_In_Kernel	239	ArchiveCheck	0.3
Sequence	20	CHR_Dev_SCH	20	GZIP	5.8
Verify	50	No_HZ	12	Adaptation	0.2
Statistic	285	NF_Conntrack_IPV6	13	Checksum	2.4
Cryptography	19	PCNET32	34	Compress*	0
Replication	89	Module_Unload	24	Extract	7
Diagnostic	191	CC_Optimize_For_Size	1,443		

* Compress is a mandatory feature

store the results in the product-line model (similar to the complexity measurement). The drawback of these measurement techniques is that they depend on the used implementation techniques and programming language.

We need a more general technique to measure non-functional properties per feature, since many SPLs are, for example, implemented by means of conditional compilation (e.g., with the C preprocessor). To measure a feature's footprint, we developed an approach that is implementation and language independent (Siegmund et al. 2011). The idea is to generate a set of variants that differ only in the presence of a single feature. The delta of the measured footprint of two variants can be interpreted as the influence of the corresponding feature on footprint. This way, we can approximate a feature's non-functional properties. The details of this approach are outside the scope of this paper and can be found in (Siegmund et al. 2011).

We show the approximated footprint of selected features of Berkeley DB, Linux kernel and ZipMe in Table 3. We refer the interested reader to our website for the measurements of the other SPLs. We can see that our approach is applicable to large SPLs (e.g., Linux with a size between 11 and 13 MB), medium SPLs (e.g., Berkeley DB's footprint range for the static library is between 1.8 and 2.7 MB) and small SPLs (e.g., for ZipMe[13] the range within 79 and 99 KB).

5.4 Measuring performance (variant-wise quantifiable property)

Often, stakeholders want to derive a performance-optimized variant. To measure performance, we have to execute the variant. That is, we have to configure the SPL, compile the variant and finally run a benchmark. Obviously, we can only measure the performance of the whole variant, not of individual features. Thus, we classify performance as a variant-wise quantifiable property. Each application domain or even each program has special

[13] Since feature Compress is a mandatory feature, it is present in every product. Hence, the size of this feature is measured together with the size of the SPL's core feature Base.

demands for measurements of runtime properties, such as performance. For instance, we measure the time for sorting of the LinkedList SPL and we use Oracle's standard read benchmark for Berkeley DB.

Since the measurement of variant-wise properties is the last phase in the variant-derivation process, we compare the results of different variants according to an objective function. Depending on the feature selection, we can observe largely differing results. We benchmarked three different variants of Berkeley DB. The variants differ in the features B-Tree, Hash and Cryptography. In our case, we use 40 runs for each benchmark to reduce the effect of measurement bias. As a result, the variant with feature B-Tree index has the best performance with respect to the workload of Oracle's standard benchmark. In average, we measured a performance of about 110,000 T/s (transactions per second). If we change the index to use the feature Hash, the performance degrades to about 45,000 T/s. Finally, we also measured the influence of feature Cryptography on performance. Not surprisingly, we found a substantial performance degradation if data are encrypted. In this case, Berkeley DB was only able to perform 2,640 T/s, which is about 43 times slower than without feature Cryptography. In SPL Conqueror, we use the aggregated result of such a benchmark in the objective function to identify an optimal variant.

5.5 Discussion

In this section, we provide details about effort and accuracy of our evaluations. In particular, we show how much time we needed to perform the measurements using SPL Conqueror compared to manual measurement (which we did in previous work). Furthermore, we evaluate the accuracy of the measurements of feature-wise quantifiable properties.

5.5.1 Time for measurements

Compared to previous work (Siegmund et al. 2008b), SPL Conqueror has an automated measurement process. SPL Conqueror does not require any user interaction (e.g., the measurement process can run over night). It automatically generates variants and applies predefined measurements to them, which significantly reduces effort for measurement. Overall, the definition of appropriate measurements did not require any domain knowledge.

Before measurement, a stakeholder (usually the vendor) has to define a metric or a program that can be applied to measure a variant or piece of source code (e.g., as we did for Source Monitor). Additionally, an user must define how SPL Conqueror can extract the results of the measurement. For example, we use *XPath* expressions to extract results from XML files. Given such a setup, we can run SPL Conqueror without any user interactions in our case.

Moreover, we could reuse the measurement setup (i.e., the definition how a non-functional property can be measured) for different SPLs, which further reduce the effort when new SPLs have to be measured. When measuring the footprint of an SPL, the largest amount of time was dedicated to the compilation process. For example, measuring footprint for the selected 25 features of the Linux kernel took us 4 days with a standard desktop computer. In Table 4, we show the average time needed to measure a feature's footprint for all SPLs. We required the most time for SPLs with either a large number of features (e.g., SQLite or Violet) or a large code base (in the case of the Linux kernel). Having a large code base increases the compilation time substantially and having a large number of

Table 4 Time spent for measuring footprint per feature of a number of variants

SPL	Measurement time	# Measured variants	Total # of variants
LinkedList	15 min	13	492
Prevayler	7 min	7	24
ZipMe	8 min	10	104
PKJab	7 min	8	72
Sensornetwork	12 h	34	3,240
Violet	24 h	2,115	ca. 10^{20}
Berkeley DB	11 h	15	256
SQLite	48 h	146	10^{23}
Linux kernel	96 h	207	ca. 33×10^{6}

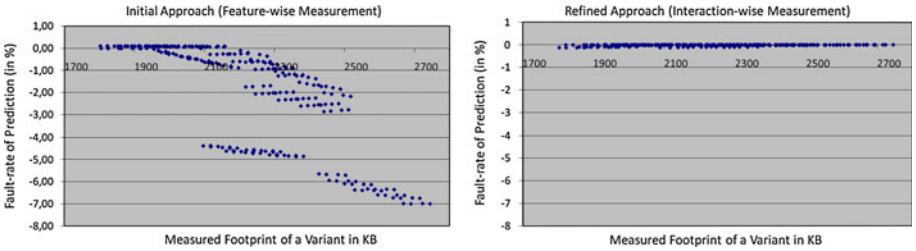

Fig. 8 Fault rates of predicted footprints of all variants of Berkeley DB using two different measurement approaches

features requires to generate, compile, and measure many variants which also increases the measurement time.

Measuring complexity took only 30 minfor all SPLs. We had to analyze the output format of Source Monitor and defined an appropriate XPath expression to extract the correct value. After we did this for one SPL, we could export the definition in SPL Conqueror and import it for all other SPLs. The definition of footprint and performance measurements took about 5 min, on average, per SPL. Depending on the existence of a makefile for building and benchmarking, we had to extract only the binary size or benchmark results either from the command line output or extract from a self-written tool. For example, we wrote a simple tool to compute the size of all class files, lib files, etc. Then, we only had to specify in SPL Conqueror how to start the tool and from which file to read the (XML-based) output. This took a minute. Again, we could reuse the tool for all SPLs.

5.5.2 Accuracy of property prediction

When predicting non-functional properties of a variant using a feature-wise measurement, we found that predictions can be inaccurate due to unknown feature interactions or compiler optimizations. In the following, we discuss our observations for the property footprint and analyze how the accuracy can be improved.

In case of Berkeley DB, we identified some minor inaccuracies for our footprint prediction.[14] On the left side of Fig. 8, we depict the fault rate of our initial predictions. One can see that we have an increasing fault rate for larger variants. The worst fault rate is 7%,

[14] We used the Microsoft C compiler and /O2 optimization level as compiler flag.

and the average fault rate is 1.9%. The reason for our inaccurate predictions is feature interactions at the source code level, such as nested #ifdef statements (Kästner et al. 2009). That is, some code fragments are only active if two or more interacting features are used at the same time. When measuring a single feature's footprint, we could not measure the influence of code fragments that are only present in a product for a certain feature combination. Hence, we refined our measurement approach to also measure combinations of features. The measured values are assigned to the derivatives in our product-line model. The existing feature interactions at source code level were easy to identify, because we only had to look for nested code fragments.

With the refined approach, we could significantly improve our predictions up to a worst-case fault rate of 0.1%. That is, we predicted the footprint of nearly every variant correctly based on a feature's footprint. The feature-wise measurement is usually accurate and has a very low complexity. In Fig. 8, we show the improved predictions for Berkeley DB on the right side. A complete description of the refined measurement approach is outside the scope of this paper and given elsewhere (Siegmund et al. 2011).

6 Computing an optimal variant

The variant-derivation process of SPL Conqueror integrates the measurement of properties, the manual selection of features by customers and the computation of an optimal feature selection based on an user-defined objective function. An objective function can be defined over multiple properties of the feature-wise and variant-wise quantifiable properties. Additionally, if a customer or a domain expert provides a mapping from a qualitative description of a property to real numbers, also qualitative properties can be used in an objective function. However, it is the responsibility of the stakeholder who provides such a mapping that the objective function produces meaningful results. In this work, we do not address the problems of defining appropriate objective functions and so refer the interested reader to according literature (Karlsson et al. 1998; Bagnall et al. 2001; Saliu and Ruhe 2007; Zhang et al. 2007). Hence, our work is orthogonal to previous work in this area and we can integrate it. For simplicity, in SPL Conqueror, we currently use a single (weighted) objective function that can be entered in a text field.

During the variant-derivation process, we face two major challenges. First, due to the large variant space, the computation of the optimal variant is very time consuming. The underlying problem is NP-hard (White et al. 2009). Second, properties of the third category (variant-wise quantifiable properties) require the generation of a variant and usually the execution of a benchmark, which requires additionally a large amount of time. Hence, we need a solution that measures only variants that are likely to be the optimal variant. To this end, we propose a staged product-derivation process, as illustrated in Fig. 9. The underlying algorithm consists of four steps: (a) feature selection to satisfy functional requirements, (b) constraining non-functional properties to reduce the search space to find an optimal variant, (c) computing a feature selection to optimize non-functional properties and (d) applying post-derivation optimizations to a derived variant. In the following, we describe each step in detail.

6.1 Feature selection

The variant derivation starts with the selection of features according to the functional requirements, (in Fig. 9a). For example, the product-line model of the Java version of the

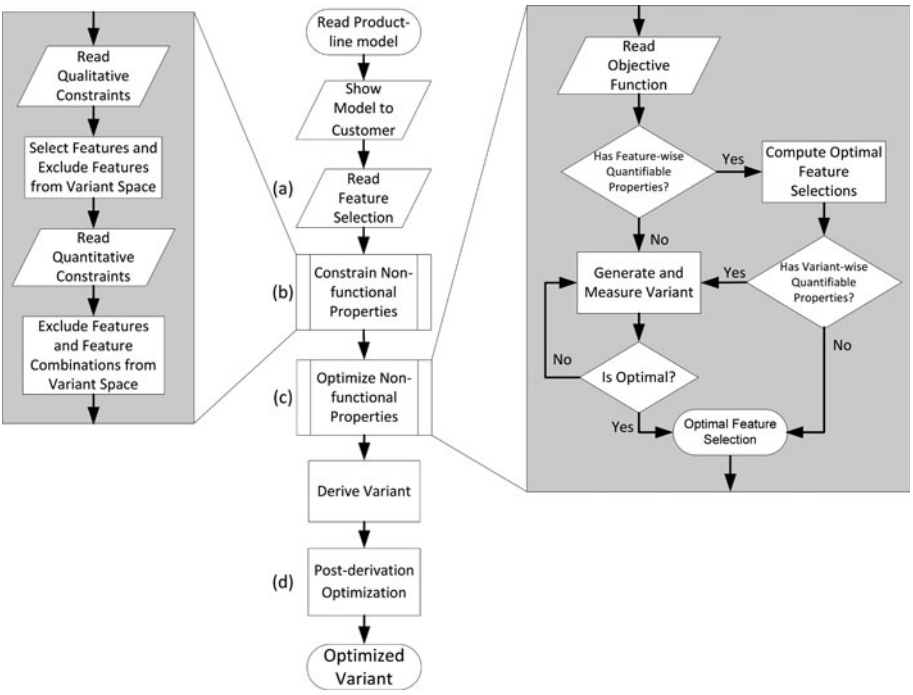

Fig. 9 Algorithm of SPL Conqueror's variant-derivation process

Berkeley DB SPL (on the left side of Fig. 10) can be used to select required features. After selecting the desired features, users can verify the correctness of the selection (e.g., find out whether there are domain constraints that require the selection of another feature). As a result of this stage, we can exclude many variants that cannot satisfy functional requirements.

6.2 Constraining non-functional properties

The aim of the second step is to exclude as many features and feature combinations as possible from the search space of an optimal variant. To this end, we use multiple techniques to define constraints for non-functional properties. For qualitative properties, we highlight features that improve or degrade the respective property. For example, we can highlight the features Replication, Verification and Diagnostic for non-functional property reliability, since a domain expert already associated these features with the property. To reduce the number of variants, we exclude features from further consideration if they have a negative effect on a property that is of interest to a customer. Although this is an approximation, it is often necessary to reduce the optimization complexity. Additionally, constraints can be defined to exclude features. For example, a customer may define a constraint that states that a DBMS variant has to be at least *medium secure* (e.g., like it is used in Windows 7). Hence, we do not have to consider weaker security mechanisms anymore.

If a customer is interested in a property that is not already assigned, either the SPL vendor or a domain expert has to perform the assignment task. Since features are usually

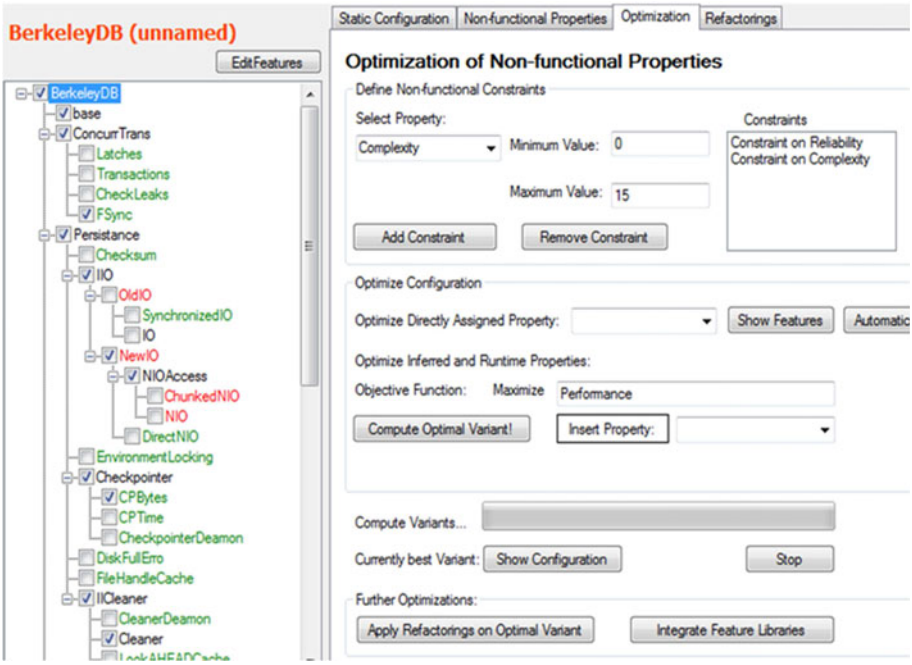

Fig. 10 Variant derivation in SPL Conqueror including constraint specification and optimization

well documented, this task is usually fast and easy to accomplish. However, it is important that the understanding of the nature of a non-functional property is consistent between SPL vendor and customer. It might be the case that an assignment of a non-functional property looks optimal for one stakeholder might not look optimal to another stakeholder. For instance, reliability can be interpreted in different ways. Thus, we store a description of the property to give rational about how the influence of features is qualitatively interpreted.

In addition to qualitative properties, a customer can define constraints for quantifiable properties in SPL Conqueror (top of Fig. 10) to reduce the search space for an optimal variant. Based on the stored non-functional properties of features, we can compute in advance whether the selection (or a certain feature combination) violates the given non-functional constraints. Then, we remove these features and feature combinations from the search space that would always violate the given non-functional constraints. If many non-functional constraints are defined with contradicting goals, it might be the case that we exclude all features or could not give any valid configuration. If so, we can give a warning to the user that there is no product that can satisfy all given constraints.

To give a concrete example for footprint, we measured the footprint of all Berkeley DB features and stored the results in the product-line model. If a customer wants to derive a variant with a footprint limit of 500 KB, we can exclude features that alone are larger than 500 KB. For instance, we have to use the small B-Tree implementation with 340 KB instead of the fast implementation (with 1,800 KB). Since feature B-Tree is a mandatory feature, we have at minimum 340 KB for a variant. Hence, we can further exclude features from the search space (e.g., feature Diagnostic) that would introduce more than 160 KB, because it would always violate the given constraint. We can also define constraints for

variant-wise quantifiable properties. However, we cannot use such constraints to exclude variants until we generate and measure a variant, which we also do in the optimization step. We would have the effort of determining variant-wise properties twice. Therefore, we postpone the Verification of such constraints to the optimization stage.

6.3 Optimization of non-functional properties

The next step in the derivation process is the computation of an optimal feature selection based on an user-defined objective function (center of Fig. 10). As an example, consider the following objective function in which a customer of Berkeley DB is interested in deriving a variant with the best trade-off between high performance and low footprint:

$$\max(\mathit{performance}/(1000 \times \mathit{footprint})) \tag{1}$$

This objective function consists of a weighted feature-wise quantifiable property and a variant-wise quantifiable property. According to our algorithm in Fig. 9, if the objective function contains only feature-wise quantifiable properties, we directly compute an optimal variant using a CSP solver. Since the computation is NP-hard, we may be able to give only an approximately good feature selection in a suitable amount of time for a large number of features. It is also possible to optimize a qualitative property with the same algorithm. To this end, a domain expert must provide a mapping from qualitative assignments to numbers, which we can use in an objective function. Since we have all features assigned with a value (zero if there is no influence), we do not have to measure a single variant. This shows that our approach scales also for this type of property. However, the usual approach to optimize a qualitative property would be to (automatically) select features that are marked as positively influencing the respective property.

Our exemplary objective function is defined additionally over a variant-wise non-functional property. In such a case, we can first compute a set of possible optimal variants based on the part of the objective function that contains only feature-wise quantifiable properties. The size of this set can be defined by a customer or SPL vendor to adjust the processing time. Then, we order this set of feature selections based on the intermediate results of the objective function and start the process of generating and measuring each remaining variant until we found the optimal variant or the process is aborted. This process is also performed only if variant-wise quantifiable properties exist in the objective function. Regarding scalability of the number of different non-functional properties in an objective function, we need only a linear number of additional measurements with respect to the number of different properties. That is, we already generate a variant of the SPL for a single variant-wise quantifiable property to compute its values for the objective function. We can use the same variant to measure all further defined non-functional properties in the objective function. Hence, our approach scales linearly with the number of defined properties in the objective function.

6.4 Post-derivation optimization

In a last step, we can apply further optimizations to the generated source code of the derived variant to improve non-functional properties. There are lots of optimization possibilities in the literature such as instruction reordering (Tiwari et al. 1994; Li and Henkel 2002), code transformations (Fei et al. 2007) or special compilers (Cooper et al. 2002) that target different non-functional properties.

Also, post-derivation optimizations specific to SPLs exist. Products of an SPL are usually generated by a set of implementation units. The generated source code can be hard to read and to maintain, and the generation process might even produce unoptimized code with respect to performance or footprint. To overcome these drawbacks, we developed a technique to further optimize a derived variant by means of refactorings. Altering the structure of a derived variant may influence certain non-functional properties. In previous work, we classified refactorings based on the influence on different non-functional properties (Siegmund et al. 2010a). For example, we can improve the execution time of a variant by applying the *Inline Method* refactoring to a method of a variant. The refactoring replaces the method call with the body of the called method and thus avoids the execution overhead of the method call. Depending on the application scenario and a careful use of this refactoring, we can improve performance by up to 50% (Götz and Pukall 2009). Since other refactorings, such as *Replace Inheritance with Composition* or *Inline Class* improve other non-functional properties (e.g., footprint or working memory consumption), we apply only refactorings to a variant that actually optimize a desired non-functional property. A detail discussion is out of scope of this paper (Siegmund et al. 2010a, b).

7 Related work

Our approach is orthogonal to other requirements and quality-engineering approaches as well as to measurement and optimization techniques. We present a holistic approach that integrates and makes use of existing measurement and optimization techniques. In the following, we describe how different approaches integrate with SPL Conqueror.

7.1 Quality models

There is a number of quality models and definitions of non-functional requirements (or properties) in the literature, see (Glinz 2007; Chung and do Prado Leite 2009) for an exhaustive survey, such as McCall's quality model (Mccall et al. 1977), Boehm's quality model (Boehm et al. 1978), SQUID, Software Quality in the Development Process, (Bøegh et al. 1999) and the ISO 9126 quality model (ISO 2001). All these models can be used by domain experts, SPL vendors or even customers to specify non-functional properties. However, they do not consider the specifics of SPL engineering (i.e., the separation of domain and application engineering) and of the variant-derivation process (i.e., the large variant space). Nevertheless, we can use the modeling of non-functional properties to evaluate whether a property is a qualitative or quantitative property. These models are orthogonal to our approach, and an integration in SPL Conqueror is promising to define proper measurements and to improve the optimization of non-functional properties.

Prometheus is an approach to model and predict non-functional properties in products of SPLs (Trendowicz and Punter 2003). It concentrates on the design and development phase. That is, Prometheus is limited to SPLs targeting a very restricted application scenario. The goal of Prometheus is to reuse measurements and definitions of non-functional properties for other product lines. But, it is not clear how it can be used for SPLs that have a broad scope with contradicting requirements depending on the application scenario. In contrast to Prometheus, we concentrate on product derivation. That is, we aim at optimizing varying non-functional properties of an SPL's product.

7.2 Measurement and prediction of non-functional properties in SPLs

There are many measurement techniques to predict a software's quality attributes (see (Rana et al. 2007) for an overview and (Lincke et al. 2010) for a comparison of selected models). However, prediction models usually target only a single property, such as reliability (Khoshgoftaar and Seliya 2003) and do not consider a variable set of assets as it would be necessary in SPL engineering. We do not propose novel measurements or prediction models, but aim at using existing ones in our approach. We can, for example, integrate source code-based measurements in SPL Conqueror (e.g., number of implemented interfaces, number of inner classes, etc., as used by (Pizzi et al. 2002)) and use existing methods to aggregate and reason about the results.

Only a few approaches apply measurements of non-functional properties to SPLs. Zubrow and Chastek proposed measures that evaluate the development effort for an SPL (Zubrow and Chastek 2003). Lopez-Herrejon and Apel express with their metrics the complexity of an SPL in terms of variation points (Lopez-Herrejon and Apel 2007) and cohesion (Apel and Beyer 2011). An approach close to our work is the measurement of the binary size of an aspect-oriented SPL (Hunleth and Cytron 2002). The authors compiled aspects in distinct files and measured the binary size. The footprint of different variants can then be computed. However, the approach does not consider other non-functional properties or the computation of an optimal variant.

Sincero et al (2007, 2010) propose to estimate a product's non-functional properties based on a knowledge base consisting of measurement results of already created variants. Using a machine learning approach, their aim is to find a correlation between feature selection and measurement. This way, they can infer how a feature influences a non-functional property during configuration. In contrast to our approach, they do not measure a feature's non-functional properties but a quantification of how a feature affects a property. During product derivation, they do not present an expected value for a product's properties, as we do, but can show with a slider how much a feature selection improves a property such as performance or not. Furthermore, they do not address the different types of non-functional properties (i.e., qualitative properties) nor they define a holistic product derivation process.

In a parallel line of research, we developed an approach to approximate non-functional properties of features (Siegmund et al. 2011). We use the measurement delta of two variants that differ only in the selection of a single feature. This delta is interpreted as the influence of the according feature on the measured non-functional property. We developed an algorithm to minimize the number of required measurements and to account for feature interactions. In contrast here, we focus on the complete product derivation process rather than only the measurement of products. We do not propose measurement techniques in this paper, but use existing techniques in SPL Conqueror. The measurement of features is only a single step toward the derivation of an optimal variant.

7.3 Variant derivation approaches

There are a number of approaches that target the development of programs with desired non-functional properties. These approaches, such as the non-functional requirement framework (Chung et al. 1999), i* framework (Yu 1997) and KAOS (van Lamsweerde 2001), are originally intended to help developers with design decisions to develop a software considering non-functional requirements. In SPL engineering, the software artifacts are usually already implemented when new customers derive a variant, but decisions

regarding desired non-functional properties can be made during the variant-derivation process. Hence, these frameworks may be suitable for an integration in SPL Conqueror, such that a goal-oriented model can be defined for an SPL's feature model.

The vast majority of variant-derivation tools focuses on reducing the complexity of the configuration process and supporting the user with advanced user interfaces during feature selection (Batory 2005; Antkiewicz and Czarnecki 2004; Czarnecki et al. 2004; Botterweck et al. 2007; Rabiser et al. 2007). These tools often use SAT solvers or Prolog (e.g., in pure::variants (Pure-systems GmbH 2004)) to verify a configuration against the constraints of the SPL.

As we explained before, we use a CSP solver to compute an optimal variant. There are also some approaches that allow an user to optimize the feature selection with regard to a specific non-functional property. Benavides et al. presented a technique based on CSP solvers to find an optimal variant (Benavides et al. 2005, 2007). The solver evaluates values attached to features in the feature model and then computes an optimal configuration for a small number of features. Unfortunately, their studies show that with an increasing number of features, the computation time exponentially grows. White et al. (2007, 2009) extended the optimized feature selection by enabling the definition of resource constraints. Moreover, they propose a solution based on filtered Cartesian flattening to approximate a nearly optimal variant for even large scale feature models. Again, we use a CSP solver in SPL Conqueror. But, both approaches might be useful in SPL Conqueror (e.g., for selecting optimal feature sets).

7.4 Optimization techniques for non-functional properties

There are a number of techniques targeting the optimization of a specific non-functional property. A related approach for optimizing non-functional properties was developed in the *COMQUAD* project (Göbel et al. 2004). The project focuses on techniques for tracing and adapting non-functional properties in component-based systems. Particularly, developers can select between alternative implementations dynamically and an infrastructure weaves these implementations as non-functional aspects in the component. This approach requires a dedicated component model based on *Enterprise JavaBeans, CORBA Components* and AOP. In contrast, SPL Conqueror is not constrained to a specific implementation technique or language. Furthermore, we consider the measurement of non-functional properties, which is not addressed in their work.

8 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. It automates the measurement of non-functional properties and derivation of optimized program variants of a product line. We allow product-line vendors to measure the features of a product line (e.g., footprint and performance) or to qualitatively rate features according to their influence on a non-functional property. By providing a classification of non-functional properties, we support different measurement techniques (feature-wise measurement and variant-wise measurement). We solve the problem of the large variant space and the large spectrum of non-functional properties in a product line by providing appropriate configuration possibilities for each class of properties embedded in a staged variant-derivation

process. We discussed an evaluation for the measurement of non-functional properties with nine case studies. The sample product lines were chosen from different domains (e.g., database and UML editor). They are implemented with different techniques and languages (C, C++, Java). This demonstrates that our approach is language, domain and implementation independent.

In future work, we will extend our approach to reduce the measurement effort for variant-wise quantifiable properties, such as performance and energy consumption. We will also work on techniques to automatically identify feature interactions at the level of non-functional properties and on an integration of other approaches for computing the optimal feature selection in SPL Conqueror. Furthermore, an important work will be the application of SPL Conqueror with an industrial setting including customer-defined non-functional requirements.

Acknowledgment We would like to thank Janet Feigenspan and the anonymous reviewers for their constructive and helpful comments which substantially improved the quality of the paper. Norbert Siegmund is funded by the German ministry of education and science BMBF, number 01IM10002B. Marko Rosenmüller is funded by the German research foundation, project number SA 465/34-1. Apel's work is supported by the DFG projects #AP 206/2-1 and #AP 206/4-1. Kästner's work is supported by the European Research Council (grand ScalPL #203099).

References

Antkiewicz, M., & Czarnecki, K. (2004). Featureplugin: Feature modeling plug-in for Eclipse. In *Workshop on eclipse technology eXchange* (pp. 67–72). New York: ACM Press.

Apel, S., & Beyer, D. (2011). Feature cohesion in software product lines: An exploratory study. In *Proceedings of the International Software Engineering Conference (ICSE)* (pp. 421–430). New York: ACM Press.

Bagnall, A. J., Rayward-Smith, V. J., & Whittley, I. M. (2001). The next release problem. *Information and Software Technology, 43*(14), 883–890.

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference (SPLC)* (Vol. 3714, pp. 7–20). Berlin: Springer, LNCS.

Batory, D., Sarvela, J. N., & Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE), 30*(6), 355–371

Benavides, D., Segura, S., Trinidad, P., & Cortés, A. R. (2007). FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)* (pp. 129–134). Berlin: Springer.

Benavides, D., Ruiz-Cortés, A., & Trinidad, P. (2005). Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering (CAISE)* (Vol. 3520, pp. 491–503). Berlin: Springer, LNCS.

Bøegh, J., Depanfilis, S., Kitchenham, B., & Pasquini, A. (1999). A method for software quality planning, control, and evaluation. *IEEE Software, 16*, 69–77.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. J., & Merritt, M. J. (1978). *Characteristics of software wuality (TRW series of software technology)*. Amsterdam: Elsevier.

Botterweck, G., Nestor, D., Preußner, A., Cawley, C., & Thiel, S. (2007). Towards supporting feature configuration by interactive visualization. In *Proceedings of Workshop on Visualisation in Software Product Line Engineering (ViSPLE), IEEE Computer Society*, pp. 125–131.

Chung, L., & do Prado Leite, J. (2009). On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications* (Vol. 5600, Chap 19, pp. 363–379). Berlin: Springer, LNCS.

Chung, L., Nixon, B. A., & Yu, E. (1995). Using non-functional requirements to systematically support change. In *Proceedings of the International Symposium on Requirements Engineering (RE), IEEE Computer Society*, pp. 132–139.

Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (1999). *Non-functional requirements in software engineering*. Berlin: Springer.

Clements, P., & Northrop, L. (2002). *Software product lines: Practices and patterns*. MA: Addison-Wesley.

Cooper, K. D., Subramanian, D., & Torczon, L. (2002). Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing, 23*(1), 7–22.

Czarnecki, K., & Eisenecker, U. (2000). *Generative programming: Methods, tools, and applications*. MA: Addison-Wesley.

Czarnecki, K., Kim, C. H. P., & Kalleberg, K. T. (2006). Feature models are views on ontologies. In *Proceedings of the International Software Product Line Conference (SPLC), IEEE Computer Society*, pp. 41–51.

Czarnecki, K., Helsen, S., & Eisenecker, U. W. (2004). Staged configuration using feature models. In *Proceedings of the International Software Product Line Conference (SPLC)* (Vol. 3154, pp. 266–283). Berlin: Springer, LNCS.

Fei, Y., Ravi, S., Raghunathan, A., & Jha, N. K. (2007). Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Transaction on Embedded Computer Systems, 7*(1), 1–26.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., & Gjorven, E. (2006). Using architecture models for runtime adaptability. *IEEE Software, 23*, 62–70.

Glinz, M. (2007). On non-functional requirements. In *Proceedings of the International Conference on Requirements Engineering (RE), IEEE Computer Society*, pp. 21–26.

Göbel, S., Pohl, C., Röttger, S., & Zschaler, S. (2004). The COMQUAD component model: Enabling dynamic selection of implementations by weaving non-functional aspects. In *International Conference on Aspect-oriented Software Development (AOSD)* (pp. 74–82). New York: ACM Press.

Götz, S., & Pukall, M. (2009). On performance of delegation in Java. In *Proceedings of the International Workshop on Hot Topics in Software Upgrades (HotSWUp)* (pp. 1–6). New York: ACM Press.

Hunleth, F., & Cytron, R. (2002). Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES)* (pp. 38–45). New York: ACM Press.

International Organization for Standardization (ISO) (2001). Software engineering—Product quality, Part 1: Quality model. In *JTC 1/SC 7—Software and systems engineering, ISO/IEC 9126-1*.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

Karlsson, J., Wohlin, C., & Regnell, B. (1998). An evaluation of methods for prioritizing software requirements. *Information and Software Technology, 39*(14–15), 939–947.

Kästner, C., Apel, S., ur Rahman, S. S., Rosenmüller, M., Batory, D. & Saake, G. (2009). On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the International Software Product Line Conference (SPLC), Software Engineering Institute (SEI)*, pp. 181–190.

Khoshgoftaar, T. M., & Seliya, N. (2003). Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering, 8*, 255–283.

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

Li, Y., & Henkel, J. (2002). *A framework for estimating and minimizing energy dissipation of embedded HW/SW systems* (pp. 259–264). Dordrecht: Kluwer Academic Publishers.

Lincke, R., Gutzmann, T., & Löwe, W. (2010). Software quality prediction models compared. In *International Conference on Quality Software (ISCQ), IEEE Computer Society*, pp. 82–91.

Lopez-Herrejon, R., & Apel, S. (2007). Measuring and characterizing crosscutting in aspect-based programs: Basic metrics and case studies. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)* (pp. 423–437). Berlin: Springer

Marler, R., & Arora, J. (2004). Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization, 26*(6), 369–395.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.

Mccall, J. A., Richards, P. K., & Walters, G. F. (1977). Factors in software quality. Vol. 1. Concepts and definitions of software quality. Technical Report ADA049014, General Electric Co Sunnyvale California.

Oracle (2006). Oracle press release. http://www.oracle.com/corporate/press/2006_sep/oracle_bdb_4-5.htm

Pizzi, N. J., Summers, R., & Pedrycz, W. (2002). Software quality prediction using median-adjusted class labels. In *International Joint Conference on Neural Networks (IJCNN), IEEE Computer Society*, pp. 2405–2409.

Pohl, K., Böckle, G., & van der Linden, F. (2005). *Software product line engineering: Foundations, principles and techniques*. Berlin: Springer.

Pure-systems GmbH (2004). Technical white paper: Variant management with pure::variants. [Avaiblable online at: http://www.pure-systems.com.

Rabiser, R., Dhungana, D., & Grünbacher, P. (2007). Tool support for product derivation in large-scale product lines: A wizard-based approach. In *Workshop on Visualisation in Software Product Line Engineering (ViSPLE), IEEE Computer Society*, pp. 119–124.

Rana, Z. A., Shamail, S., & Awais, M. M. (2007). A survey of measurement-based software quality prediction techniques. Tech. Rep. Lahore University of Management Sciences.

Robertson, S., & Robertson, J. (1999). *Mastering the requirements process*. New York: ACM Press.

Saliu, M. O., & Ruhe, G. (2007). Bi-objective release planning for evolving software systems. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)* (pp. 105–114). New York: ACM Press, FSE.

Siegmund, N., Kuhlemann, M., Rosenmüller, M., Kästner, C., & Saake, G. (2008a). Integrated product line model for semi-automated product derivation using non-functional properties. In *Workshop on Variability Modelling of Software-intensive Systems (VaMoS)* (pp. 25–31). University of Duisburg-Essen.

Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., & Saake, G. (2008b). Measuring non-functional properties in software product lines for product derivation. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society*, pp. 187–194.

Siegmund, N., Kuhlemann, M., Apel, S., & Pukall, M. (2010a). Optimizing non-functional properties of software product lines by means of refactorings. In *Proceedings of Workshop Variability Modelling of Software-intensive Systems (VaMoS)* (pp. 115–122). University of Duisburg-Essen.

Siegmund, N., Rosenmüller, M., & Apel, S. (2010b). Automating energy optimization with features. In *Proceedings of International Workshop on Feature-oriented Software Development (FOSD)* (pp. 2–9). New York: ACM Press.

Siegmund, N., Rosenmüller, M., Kästner, C., Giarusso, P. G., Apel, S., & Kolesnikov, S. S. (2011). Scalable prediction of non-functional properties in software product lines. In *Software Product Line Conference (SPLC), IEEE Computer Society*.

Sincero, J., Schröder-Preikschat, W., & Spinczyk, O. (2010). Approaching non-functional properties of software product lines: Learning from products. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society*, pp. 147–155.

Sincero, J., Spinczyk, O., & Schröder-Preikschat, W. (2007). On the configuration of non-functional properties in software product lines. In *Software Product Line Conference (SPLC), Doctoral Symposium* (pp. 167–173). Kindai Kagaku Sha Co. Ltd.

SQLite.org (2010). Press release. http://www.sqlite.org/mostdeployed.html [Accessed at: 19th May 2011].

Stevens, S. S. (1946). On the theory of scales of measurement. *Sciences, 103*(2684), 677–680.

Tiwari, V., Malik, S., & Wolfe, A. (1994). Compilation techniques for low energy: An overview. In *Proceedings of Symposium on Low Power Electronics (ISLPED), IEEE Computer Society*, pp. 38–39.

Trendowicz, A., & Punter, T. (2003). Quality modeling for software product lines. In *ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*.

van Lamsweerde, A. (2001). Goal-oriented requirements engineering: a guided tour. In *International Symposium on Requirements Engineering (RE), IEEE Computer Society*, pp. 249–262.

van Lamsweerde, A., Darimont, R., & Letier, E. (1998). Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering, 24*(11), 908 –926.

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

White, J., Dougherty, B., & Schmidt, D. C. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software, 82*(8), 1268–1284.

Yu, E. S. K. (1997). Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the International Symposium on Requirements Engineering (RE), IEEE Computer Society*, pp. 226–235.

Zhang, Y., Harman, M., & Mansouri, S. A. (2007). The multi-objective next release problem. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO)* (pp. 1129–1137). New York: ACM Press.

Zubrow, D., & Chastek, G. (2003). Measures for software product lines. Tech. Rep. CMU/SEI-2003-TN-031, Carnegie Mellon University.

Author Biographies

Norbert Siegmund received his Master in Computer Science (degree: Diplom-Informatiker) from Otto-von-Guericke University, Magdeburg, Germany in 2007. He joined immediately the database workgroup at the Otto-von- Guericke University in Magdeburg as a PhD student. His research interests are software product line engineering techniques and non-functional properties.

Marko Rosenmüller received his Diploma in Computer Science from the University of Magdeburg, Germany in 2005. From 2000 to 2006, he was a software developer at the icubic AG in Magdeburg. Since 2006, he is a Ph.D. student at the University of Magdeburg. His research interests include software product lines, tailor-made data management and programming languages for product line development.

Martin Kuhlemann received his Master degree in Computer Science (Diplom-Informatiker) at the University of Magdeburg, Germany, in 2006. After that, he joined the database research group of the faculty of computer science at the Univerisity of Magdeburg. His research interests include software product lines, refactoring and generative programming.

Christian Kästner is a post-doctoral at Philipps University Marburg in Germany. He received a Ph.D. in Computer Science from the University of Magdeburg, Germany in 2010. His research interests include languages and tools for software product lines and (virtual) separation of concerns.

Sven Apel is a post-doctoral associate at the Chair of Programming at the University of Passau, Germany. He received a Ph.D. in Computer Science from the University of Magdeburg, Germany in 2007. His research interests include advanced programming paradigms, software product lines and algebra for software construction.

Gunter Saake received the diploma and a PhD in Computer Science from the Technical University of Braunschweig, F.R.G. in 1985 and 1988, respectively. From 1988 to 1989, he was a visiting scientist at the IBM Heidelberg Scientific Center, where he joined the Advanced Information Management project and worked on language features and algorithms for sorting and duplicate elimination in nested relational database structures. In January 1993, he received the Habilitation degree (venia legendi) for Computer Science from the Technical University of Braunschweig. Since May 1994, Gunter Saake is a fulltime professor for the area "Databases and Information Systems" at the Otto-von-Guericke University, Magdeburg. His research interests include database integration, tailor-made data management, object-oriented information systems and information fusion.