On the Configuration of Non-Functional Properties in Software Product Lines

Julio Sincero, Olaf Spinczyk, Wolfgang Schröder-Preikschat
Friedrich-Alexander University of Erlangen-Nuremberg
Department of Computer Science
Martensstr. 1, D-91058 Erlangen, Germany
{sincero,os,wosch}@cs.fau.de

Abstract

The software product line (SPL) practices intend to improve software development by automating the process of product derivation. However, little attention has been given to the configuration of non-functional properties in the context of SPL infrastructure. To address this shortcoming we introduce the Feedback *approach that aims at capturing information about previously configured products and re-inserting it back into the SPL infrastructure. We describe how information of configured products could be logically organized and used by the SPL infrastructure to facilitate the configuration of non-functional properties, and consequently, improve the configuration process of further products.*

1 Introduction

The purpose of most of the software engineering techniques is to reduce the development efforts and consequently drop costs. Software product line practices pursue this goal by providing the guidelines for the development of components that can be combined together in order to enable the automatic generation of specific products.

For the success of a SPL, the commonalities and variabilities of the target domain must be carefully identified. Domain engineering is the activity responsible for gathering this information and producing a consistent documentation of the domain that will be used in the next steps of the SPL development. The key result of the domain analysis is the set of features that the SPL will implement, and therefore, determine the user configuration choices.

The main goal of this work is to achieve a higher degree of configurability by approaching non-functional properties (NFPs) in the context of SPLs. Many types of NFPs can be better identified during software utilization [17] and cannot be well encapsulated in any software entity [15], for example, *performance, reliability, dependability*, etc. Our experience with the development of families of operating systems [10, 16] shows that the characteristics of these properties emerge from the complex interactions among the modules that comprise the entire system.

The SPL infrastructure is responsible for a variety of activities. Among them, the assembling of the components that will comprise the required product. Hence, at the configuration stage the SPL is aware of the interface of each component and how they can be assembled together. Therefore, the investigation of NFP in connection with SPL practices seems to be very promising, as the SPL infrastructure is responsible for reasoning how to combine the required components.

2 Software Product Line Engineering

The development of a family of systems that share a set of core assets is known as *Software Product Line Engineering (SPLE).* The guidelines of this technique provide methods to identify commonalities and variabilities in a desired domain and to develop the corresponding components that can be assembled together in order to produce a member of the family of systems (product generation).

For the effectiveness of a SPL the domain must be carefully studied and documented so that the results can be used in the next phases of development. *Domain engineering* is normally employed for the domain analysis. It results in a comprehensive model that represents the domain emphasizing the common and the variable. *Feature modeling*[12] is a technique that per-

mits the description of a domain by means of *mandatory, optional, alternative* and *or* features that can be used during domain engineering in order to formally describe the variability of the domain. This document is used in the subsequent phases of the SPLE.

The *domain design* produces the software architecture for the SPL, it must cover all the features previously identified and be flexible enough to adapt itself to the variation points of the domain. The variability can be handled by a myriad of mechanisms, among them are[11]: aggregation/delegation, inheritance, parametrization, overloading, libraries, conditional compilation, AOP (aspect-oriented programming), design patterns, etc.

Domain implementation is the actual coding of the components described by the software architecture. The resulting components should be able to be either automatically or manually combined in order to yield a specific product.

2.1 Infrastructure Product Lines

The SPLE guidelines do not impose restrictions to any type of software. Many study cases have shown successful product lines in a variety of domains, like embedded systems [13], middleware [6], operating systems [14], etc.

Infrastructure *SPLs* represent the type of SPL where its member products are meant to provide services to software entities in a higher level of abstraction, examples are SPLs that generate products like operating systems, data bases, middlewares, etc.

We think that during product derivation, the *SPL user* (application engineer) should be aware of the price of each and every feature that is being selected and will later incorporate the final product. However, this characteristic is of extreme importance in the case of *infrastructure SPLs,* as it still has to be combined to applications that require its services, and also, the system's resources have to be shared among the different components that comprise the entire application, as a result, the demand of each of them must be known so that the application engineer can evaluate its design decisions.

Imagine the scenario where the resulting products of two complementary SPLs have to combined in order to form a final product. The first is a operating for an specific embedded platform with constrained memory resources. The second, an user application that is able to use the interface provided by the operating system. During the configuration process of the application, it must be known how much of the system's resources are still left for the application, it means, how much is the infrastructure consuming of it. Sensible questions that may arise during the product configuration are: *how much of RAM and ROM memory is available for the application?* In the case of an performance-critical application, *what is the memory allocation overhead imposed by the underlying operating system?*

2.2 Tool Support

Tool support can help during all phases of SPLE. Basically, in each development step a model is generated for later use, and after the development, the *use* (product generation) of the SPL is characterized by model transformation, which, in short, means the mapping from the problem domain to the solution domain.

The literature contains many examples of tools to support [2, 5] product specification. This work aims at discussing new ideas to improve tool support in the realm of *product configuration*, for example:

- During product configuration we want to inform the user how the current selection affects the non-functional properties of the product to be generated.
- Offer explicit means to select desired non-functional properties of the product, and consequently, provide to the user the set of features that should be selected/deselected for the desired NFP compliance.

In order to achieve these goals the SPL tool must be able to:

- Provide structures to store information about previously configured systems.
- Provide mechanisms to create rules using information about the *feature selection* and the information of *beforehand generated products.*

3 Product Configuration

Product configuration is the act of selecting the desired features for a specific product. Recent research in the area of product configuration has shown the complexity of this domain. Staged configuration [9] is introduced by Czarnecki, he explains a stepwise specialization of feature modules where the configuration choices of each stage are defined by separate feature models, the approach is motivated by the characteristic of realistic development process, where different groups and different people make configuration choices in different

stages. Mendonca [18] presents a process-centric approach to collaborative and coordinated product configuration, his approach attempts to anticipate decision conflicts by allowing decision makers priority schemes that will serve as basis for resolving conflicts. Schirmeier [21] believes that multi-level architectures of layered product lines will be common practice in the future, and therefore, feature-based configuration will become increasingly complex. To solve this problem, a tool for static code analysis is presented. The goal is to automate the configuration process by deriving as much as possible of the required features from an application by static analysis.

We believe in the synergy of the approaches above described and we agree that product configuration in real scenarios turns out to be one of the most complex tasks in SPLE. Although in general product configuration problems are tackled by all of them, they address different facets of the same general problem. Moreover, we understand that these approaches could be complemented by the ideas of this work to address the configuration of non-functional properties.

3.1 Non-Functional Properties Configuration

Non-functional properties are those that do not express *what* a software is able to compute, but *how,* or *in which circumstances* it achieves its main goals (the functional properties) . Typical examples of NFPs are: performance, memory footprint, platform compatibility, reliability, robustness, safety, security, etc. There are no formal definitions of any of these NFPs, different stakeholders may have different expectations on the system's NFPs.

The explicit definition of NFPs during software configuration is not a common practice. In the best scenarios the configuration tool informs the user how a specific feature *may* affect *some* NFPs. To illustrate, Table 3.1 shows excerpts of the configuration help provided by the configuration tool of the Linux Kernel [1]. As it can be seen, to express how NFPs like performance, code size and memory footprint, vary according to *feature selection*, quite vague descriptions (bold text in the table) are used.

At the first sight these enigmatic messages may seem to have an arguable usefulness. However, they represent the desire of understating the impact of *feature selection* during product derivation on NFPs. Furthermore, if these messages pass the rigorous process of clean-ups imposed by the community that collaboratively develops the Linux Kernel, we can conclude that there are stakeholders who are interested even in the most elementary types of information about NFPs as presented above.

Sysctl support: "... Enabling this option will **enlarge** the kernel by **at least 8 KB**".
Check for stack overflows: "...this option will **slow down** process creation **somewhat**".
SMT Hyperthreading: "...at a cost of **slightly** increased **overhead** in **some places**".
Memory Type Range Register: "...this can **increase performance** of image write operations **2.5 times** or more".
Voluntary Kernel Preemption (Desktop): "...providing **faster** application reactions, at the cost of **slighly lower** throughput".

Table 1. Linux Kernel Configuration Messages

4 Problem Description

The guidelines of SPLE are the evolution and combination of several software engineering techniques like *domain engineering, feature modeling, family software development, generative programming, model-driven development, object-/aspect-/feature-oriented development, etc.* Although there are methods for approaching NFP during all phases of application engineering, including requirements specification [19], modeling of NFP using UML diagrams [7], and modeling of NFP in software architectures [22], little or no attention has been given to the configuration of NFP in the context of SPL infrastructure.

One of the main goals of SPLE is to automate product generation. This is normally achieved by model transformation, that is, based on a specification (e.g. feature selection) the *SPL infrastructure* selects which components from the *core assets* must be present in the final product. In practice, the *SPL infrastructure* generates config files, typedef declarations, compilation and linkage definitions, macros, pre-processor flags, etc., so that the final product can be compiled and generate the application with the exactly functionality defined in the specification. This conventional scenario is depicted in Figure 1, the SPL infrastructure use the information about its *core assets* and *feature selection* to perform the mapping from problem to solution domain, represented by the circle, and output the definition for the build of the required product. After that, no attention is given to the generated prod-

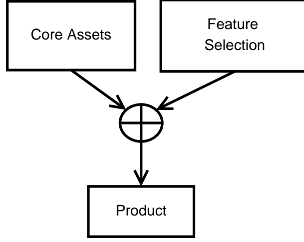

Figure 1. Conventional SPL Infrastructure

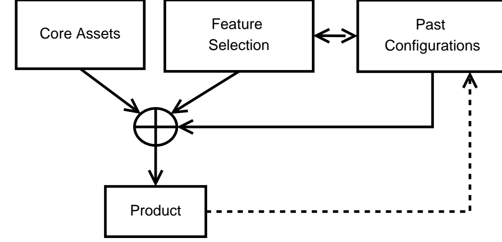

Figure 2. Extended SPL Infrastructure

uct, much relevant information about NFP, like , code size, memory footprint, etc., that could help the *SPL user* during further product configuration is simply discarded. Moreover, subsequent results of product tests, evaluation, monitoring, bug information, etc., have no means to be incorporated by the *SPL infrastructure.* Figure 2 depicts a set of extensions to the conventional *SPL infrastructure* to overcome this problem. As it can be seen, during *feature selection* the user should benefit from information of previously configured products, this is represented by the arrow connecting the boxes *feature selection* and *past configurations, additionally,* the mapping from problem to solution domain (represented by the circle) should also take advantage of the information under *past configurations.* The dotted line represents the path that the information about generated products should go through.

The basic SPL nature of automated product generation, which means domain mappings and transformations, corroborates the feasibility of the idea. Consider the scenario where a feature can be implemented by two different components, how is the *SPL infrastructure* to reason which is the best choice? This question could be answered by the different non-functional behavior of these implementation captured from previously generated products.

Therefore, we conclude that, currently, the experience of product configuration lacks support for the configuration of NFPs. The drawbacks are summarized as follows: (1) During *feature selection* the system's NFP are not taken into consideration, the *SPL user* does not know how its choices are affecting the product in terms of non-functional behavior, (2) There are no means to explicitly express the desired non-functional behavior, and then, be informed if this behavior can be achieved and what changes in the configuration are necessary.

5 The Feedback Approach

The overview of the *Feedback* approach is depicted in Figure 3. The layers *software product line repository*, *user configuration* and *concrete solution domain* represent the SPL's set of components, the user action of specifying/deriving a specific product, and the generated final product, respectively. Our goal is to provide a fine-grained level of configuration, in special the possibility to configure the system's non-functional properties. This is illustrated by the sliders on the layer *user configuration* shown in Figure 3. We want to offer to the user means to express his/her needs regarding the non-functional properties of the system:

1. We want to better assist the manual configuration, i.e. selection of features, by providing exact information about non-functional properties of the selected product variant if this variant has been configured and analyzed in the past.

2. Based on heuristics we want to provide similar information even for product variants that have not been analyzed yet.

3. If more than one system configuration on the component level can yield the same (functional) features, the user should be able to select the best variant based on the information about the expected non-functional properties.

In order to achieve these goals, the SPL infrastructure must be able to reason how each of the components being used to assemble a product impact in the properties selected by the user. Non-functional properties in most cases can not be modeled or implemented by simple software entities like classes or aspects, as they emerge from complex interactions among many modules of the running system, many non-functional properties can only be validated by performing tests which capture results at run-time.

The *Feedback* approach enables the user to select special components that do not implement the features provided by the SPL but are able to analyze the core components, as shown in the layer *user configuration* of Figure 3. The generated product can then be deployed and each of the modules comprising it can be analyzed, moreover, source and object code level analysis can be

employed as well. Examples of uncomplicated but valuable analysis are, *code size*, *memory footprint* and *execution time*. The output is generated so that it can be re-inserted in the product line and further product derivations can profit on the information captured from previous generated products. The generated information is divided in three categories (as depicted on the left side of Figure 3). *Configuration properties* stores information regarding complete product specifications. *Feature properties* represents how determined property is affected by a the selection of a feature. *Component properties* is comprised of information only relevant to components of the system.

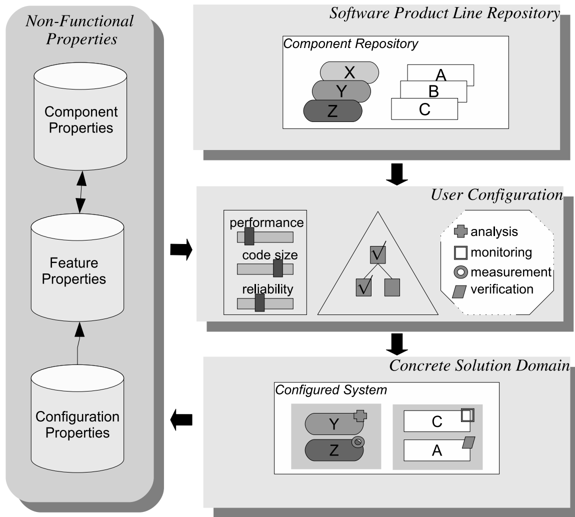

Figure 3. The Feedback Approach Overview

6 Examples

The previous section explained our approach on how to organize information of configured products in terms of *configuration*, *features* and *components properties.* In this section we want to give practical examples of each of these types of properties and exemplify how to obtain them.

To perform our tests we have used the Linux Kernel 2.6.17.1, even though this open source project was not developed using the SPLE guidelines, it achieves many of the SPLE goals, as it is able to generate different products (kernels) for a variety of platforms while providing a huge set of configurable features. In addition, we are mostly interested in working with infrastructure SPLs, where the non-functional behavior is of great interest, and the Linux Kernel offers a very large open source code base (core assets) that can be configured by a graphical tool, and as in every (or most) infrastructure software, especially kernel code, non-functional behavior can be very sensitive to, and therefore, vary, according to the selected features.

The simplest example to understand how to organize information about *configuration*, *features* and *components properties* is code size analysis. By *configuration properties* we mean information regarding a complete product specification. For example, what is the code size of kernel with the minimum possible set of features? or, What is the size of the kernel with all features selected? Of course, intermediate configurations are also examples of configurations properties. In order to obtain and save this information for further use, we had to compile the kernel with the options *allnoconfig* and *allyesconfig* to find out the respectively lower and upper bound regarding kernel image size[1]. Subsequently, we extended the Linux Kernel compilation scripts to save, after a successful kernel compilation, the configuration file (.config file, i.e. product specification), the size of the kernel image and each generated object file. Certainly, the compilation of all possible configurations to obtain the code size of all configuration is not a feasible idea. Our approach aims at progressively increase the information about *configuration properties* as different products are generated, or also, being able to perform an explicit test (during product configuration) to determine a required property.

Feature properties represent information about the selection of a specific feature during configuration. In the context of code size analysis of the Linux Kernel, it means the variation in the kernel size image by the selection of a feature. The Linux Kernel compilation system relies heavily on pre-processor macros definitions. Basically, the selection of feature in the configuration tool will reflect in the definition of a macro (a C preprocessor define) that is used in the components' code (to enable or disable the inclusion of the feature's code, it may spans to several compilation units) and also in the makefile in order to select the inclusion or exclusion of specific compile units. Our main goal is to provide a good estimate of the price of specific features during product configuration, and not to precisely predict the code size of the final product. This is not achievable, even if one measures each compilation unit separately with no features, and then again the same process for all features separately, because different compiler optimizations could take place. However, as an estimate to provide the price in terms of code size of a specific feature it is very reasonable.

Component properties relate to information about the components repository, that is, information rele-

[1] Of course any code size measure is compiler and architecture dependent.

vant to compilation units as whole, not considering the feature semantics over it.

Naturally, providing information about code size during product derivation is just one example of non-functional property configuration. We are also investigating the influence of features on non-functional properties that emerge from the complex interactions among different components. Currently, we are tackling the problem of configuring *performance* and *reliability.* For the first, the idea is to perform a *standard test* and subsequently perform the same test for different product configurations in order to find out how individual features influence on the desired non-functional behavior. For the second, the idea is to organize features that have influence on a specific non-functional property in logical groups and, if necessary, provide different *weights* to each of them. For example, the Linux Kernel offers features for deadlock detection in mutexes and spinlocks, error checking for high memory systems, detection of soft lockups, etc.. We are analysing each of these features in order to give sensible weights to each of them, and consequently, provide the user the possibility to check how reliable is the current feature selection, or explicitly set the required level of reliability so that the configuration tool can inform the necessary changes in the selection.

7 Challenges

As stated in Section 3, the process of product configuration is a complex task. The software product line infrastructure must be able to reason, based on a problem description (the product configuration), how to provide a solution (the product itself). It means to find in the *core assets* repository the required components and assemble them together. We are trying to bring even more information to this decision process, so that, during the reasoning for the generation of the required product, information about NFP of previously configured systems are also taken into consideration. Naturally, the complexity of rules for product generation tends to increase.

Regarding the result of product tests that are reinserted in the product line infrastructure, they may become invalid as the SPL components evolve, therefore, means to check the validity of this type of information must be created. This problem could be tackled by associating the SPL to a SCM (source code management) system. Then specific test results could be assigned to specific component revisions controlled by the SCM system.

8 Related Work

The On-line Repository for Embedded Software (ORES) [23] is an integrated mechanism for component-based development of embedded software. It uses an ontology-based approach to facilitate repository browsing and retrieval. Non-functional properties of components are captured and facilities for the analysis of overall system properties are provided. For the NFP analysis, the user is able to specify component instantiations and component configuration parameter values to obtain analysis results.

pure::variants [5] is a variant management system that can be used as a SPL infrastructure. It provides graphical tool support for the design feature and family models, the latter comprises the rules for the mapping from the problem to solution domain. The last release of pure::variants brings a Bugzilla connector that allows users to monitor variant-specific defect states in Bugzilla databases, this enables the user to see which variants are affected by the currently existing defects. It is also a form of tracking non-functional information in the SPL infrastructure.

There are also a series [20, 8, 7] of works in the area of modeling non-functional properties during application engineering. These approaches take into consideration required non-functional behavior and adds it to design documentation in order to make the non-functional properties traceable.

NFPs are also addressed during software architecture design [4, 3], since these techniques tackle the same problem in a different stage of development, we see our work as a complementary approach, and we believe in the synergy between them.

The area of autonomic computing aims at creating self-managing computer systems to cope with their growing complexity. It is divided in for functional areas, self-configuration, self-healing, self-optimization and self-protections. The goal of self-optimization is to monitor and to control resources to ensure optimal functioning regarding the defined requirements.

9 Conclusion

The goal of the feedback approach is to improve configuration of non-functional properties during product configuration. We discussed how to logically organize information from previous product configurations and use it in order to improve configuration of further products.

We are still in an early stage of development. In order to further develop our ideas, we are preparing

performance tests to be run under different configurations so that the impact of each desired feature on a standard test could be identified. Afterwards, this information will be incorporated into the configuration tool and heuristics to help the configuration of non-functional properties will be derived.

References

[1] linux Kernel homepage. http://www.lkml.org/.

[2] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for Eclipse. In *20 04 OOPSLA workshop on eclipse technology eXchange (eclipse '04 at OOPPSLA '04)*, pages 67–72, Vancouver, Canada, July 2004.

[3] L. Bass. Principles for designing software architecture to achieve quality attribute requirements. In *SERA*, page 2, 2006.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[5] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[6] M. L. Cecilia. Business case for a product line of legacy application data-middleware.

[7] L. M. Cysneiros and J. C. S. do Prado Leite. Non-functional requirements: From elicitation to conceptual models. *IEEE Transactions on Software Engineering*, 30(5):328–350, 2004.

[8] Y. E. L. J. Cysneiros, L.M. Cataloguing non-functional requirements as softgoals networks.

[9] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *SPLC*, pages 266–283, 2004.

[10] W. Friess, J. Sincero, and W. Schröder-Preikschat. Modelling compositions of modular embedded software product lines. In *IASTED Conf. on Software Engineering*, 2007.

[11] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *2001 Software Reusability: Putting Software Reuse in Context*, pages 109–117, 2001.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

[13] R. Kolb, I. John, J. Knodel, D. Muthig, U. Haury, and G. Meier. Experiences with product line development of embedded systems at testo ag. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 172–181, Washington, DC, USA, 2006. IEEE Computer Society.

[14] D. Lohmann, F. Scheler, W. Schröder-Preikschat, and O. Spinczyk. PURE embedded operating systems – CiAO. In *ECRTS Workshop on Operating Systems Platforms for Embedded Real-Time applications (ECRTS-OSPERT '06)*, 2006.

[15] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *10th (WORDS '05)*, pages 413–420, Sedona, AZ, USA, Feb. 2005.

[16] D. Lohmann and O. Spinczyk. Developing embedded software product lines with AspectC++. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, New York, 2006.

[17] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *4th AOSD (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).

[18] M. Mendonça, D. D. Cowan, and T. C. de Oliveira. A process-centric approach for coordinating product configuration decisions. In *HICSS*, page 283, 2007.

[19] L. Rosenhainer. The discern method: Dealing separately with crosscutting concerns.

[20] L. Rosenhainer. Identifying crosscutting concerns in requirements specifications.

[21] H. Schirmeier and O. Spinczyk. Maßschneiderung von infrastruktursoftwareproduktlinien durch statische anwendungsanalyse. In *BTW Workshops*, pages 355–365, 2007.

[22] L. Xu, H. Ziv, D. Richardson, and Z. Liu. Towards modeling non-functional requirements in software architecture.

[23] I.-L. Yen, J. Goluguri, F. B. Bastani, L. Khan, and J. Linn. A component-based approach for embedded software development. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 402–, 2002.