Is The Linux Kernel a Software Product Line?*

Julio Sincero[1], Horst Schirmeier[2], Wolfgang Schröder-Preikschat[1], and Olaf Spinczyk[2]

[1]Friedrich-Alexander University of Erlangen-Nuremberg, Germany
{sincero,wosch}@cs.fau.de
[2]University of Dortmund, Germany
{Horst.Schirmeier,Olaf.Spinczyk}@udo.edu

Abstract

The software product line (SPL) community defines SPLs not only by technical aspects like configurability and code reuse among product line variants, but also by the engineering process that accompanies the development. This paper discusses the question whether this engineering process is a mandatory property of SPLs; it concretely examines the Linux Kernel, which is not being developed according to the SPL engineering guidelines, but nevertheless shares significant commonalities with SPLs that are developed conforming to the definition.

1. Introduction

A software product line (SPL) corresponds to a set of software components that can be assembled together in order to deliver different products in a specific domain. There are several guidelines for the development of SPLs from scratch [6, 8]. Normally, the initial phase is characterized as the *domain analysis* where the *domain engineering* takes place. The commonalities and variabilities of the target domain are captured and catalogued for subsequent reuse. Based on these results, a flexible architecture that comprises the variability previously identified is specified. Finally, the *domain implementation* represents the actual coding of the components that can be configured and combined in order to generate a required product.

The development of components that are used to generate different products represents effective *software reuse.* The software engineering community has tackled this problem for decades: the SPL guidelines are proving to be successful regarding the level of *reuse* they can achieve. However, software reuse is not the only (technical) goal of SPL development; among others, improvement of time-to-market, product quality, and mass customization is expected [8].

Interestingly, another software development movement is showing good results in many areas as the SPL approach does [13, 1]. There are several well established *open source* projects that offer highly configurable software systems that achieve the same technical goals that the SPL guidelines aim at.

A good example is the Linux Kernel project [4]. It is comprised of a huge set of components that can be configured in order to generate specific kernel images (products) for a myriad of scenarios (the combination of several platforms, subsystems, device drivers, etc.). Therefore, at first sight, it seems to achieve important objectives of the SPL development like automatic product generation, code reuse, and mass customization.

The goal of this work is to study the Linux Kernel and identify if this open source project achieves the goals that the SPL community also pursues. We concentrate on the technical aspects of the SPL development like variability and asset management, product configuration and generation, and the development process. When necessary, we compare the strategies used in the Linux Kernel and the ones provided by the SPL community.

2. Motivation

Our interest in the Linux Kernel started with the necessity of a project in our research group, which investigates the configuration of non-functional properties in the con-

*This work was partly supported by the German Research Council (DFG) under grant no. SP 968/2-1.

text of operating system product lines [11]. To perform our tests, we needed an SPL with a very large code base. As we did not have the time to implement one from scratch, we decided to find an existing operating system product line with source code available, implemented using the SPL practices. Unfortunately, there was no alternative available. Therefore, we had to find an open source operating system with a high degree of configurability to play the role of an SPL in our research project. This decision raised questions like *"what is the difference between highly configurable software and SPLs?*", and, as the Linux Kernel suited our needs, *"is the Linux Kernel an SPL?*".

3. The Linux Kernel from the SPL perspective

The Linux Kernel is an open source implementation of a Unix-like operating system kernel. Currently, it supports several hardware platforms and a huge diversity of devices. It aims towards providing the complete user and software interface specified in POSIX[1] and SUS[2], and full customization in all its components, providing a very small and compact kernel[4].

The next subsections analyze the Linux Kernel with respect to the practices normally employed for SPL development.

3.1. Variability Management

In an SPL, products are comprised of a common part that is shared by the whole portfolio and a variable part that differentiates individual products. Variability can be seen as an assumption about how members of a family of products may differ from each other [12]. Also, it can be seen as the ability to change or customize a system [14].

Variability management is a discipline that is used during SPL development in order to organize and document the variability during the different stages of the life cycle. In order to understand the variability of the Linux Kernel[3], it is necessary to identify its variation points:

Hardware Architecture An operating system kernel is the closest software abstraction to the underlying hardware, the configuration of the target architecture and platform is a primordial task. The Linux Kernel offers support for more than 60 different hardware platforms (e.g. ia32, DEC, sun-4, etc.), which are organized according to the correspondent architecture (currently around 25, e.g. Sparc, Alpha, i386, etc.).

Subsystems The Kernel subsystems are organized as follows: *kernel* (architecture-independent kernel code, e.g. IRQ-handling, process scheduler, etc.), *fs* (file systems implementation), *init* (kernel initialization routines), *mm* (memory management), *sound* (sound subsystem), *block* (abstraction layer for disk access), *ipc* (inter-process communication code), *net* (network protocols), and *lib* (library functions, e.g., CRC and SHA-1 algorithms).

Device Drivers Device drivers enable the operating system to interact with a specific hardware device. This is the biggest subsystem in the Linux Kernel, in terms of lines of code; this is due to the vast number of devices it supports. For example, USB devices, network interface cards, PCMCIA cards, and a lot more.

Config Options They are responsible for configuring/-parametrizing specific features. They are spread over the whole project and can be used for selecting the required subsystems, enabling specific features of a hardware platform or device driver, and also to configure compile and debug options.

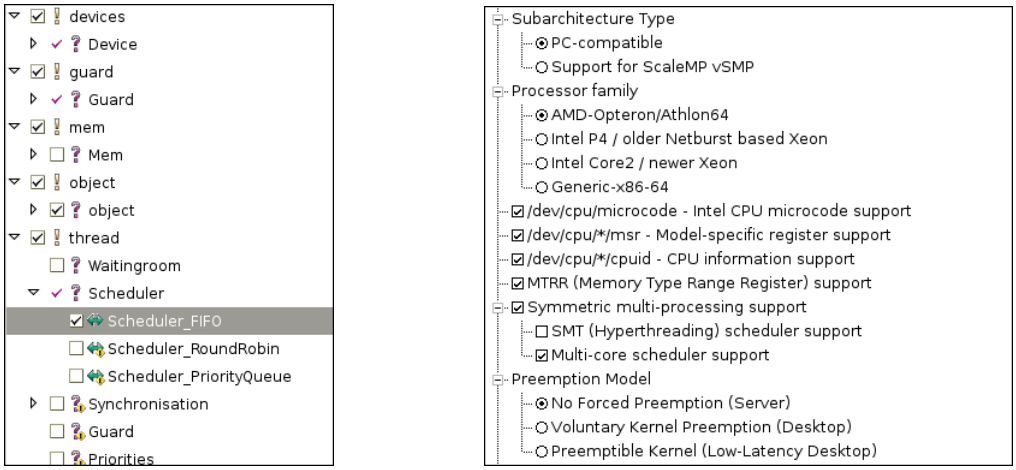
This extensive set of options represents the configurability present in the kernel. In order to generate a useful kernel image, decisions regarding the target platform, the required subsystems, and the necessary device drivers must be taken. Moreover, this selected functionality can be more finely configured by means of *Config Options*. The granularity of these choices is very inhomogeneous; for example, the choice of not including network capabilities means thousands of lines of code being left aside, whereas the selection of a debug option may result in the inclusion of few lines of code. The feature binding can occur either at runtime or compile time; in the former case the particular features must be compiled as separately loadable modules.

Regarding the management of this complex variability, the Linux Kernel uses a top-down approach. When new functionality is added to the kernel, the new configuration options are also added. The SPL community normally uses a bottom-up approach: during software design, the variation points are identified and used during the subsequent development phases.

3.2. Product Configuration

Product configuration is the process of selecting the features the desired final product is supposed to contain. The result is a list of features and parameters that is used as input for the actual generation of the product. There are automatic [10] and manual approaches for the product configuration in the context of SPLs. The Linux Kernel can be completely configured by manually editing textual configuration files. However, it offers several (equivalent) configuration tools, which only differ in their graphical interface;

[1] Portable Operating System Interface
[2] Single UNIX Specification
[3] We used the 2.6.21 release for our analysis.



(a) pure::variants configuration GUI

(b) Linux Kernel configuration GUI

Figure 1. Different configuration tools

there exist line-oriented, menu-based, and also Qt and GTK configuration front-ends.

The appearance and semantics of these configuration front-ends is very similar to modern feature model based configuration tools. On the left side, Figure 1 shows an example of a feature model[4] taken from the pure::variants [2] SPL development tool. On the right side of Figure 1, the Qt front-end for the configuration of the Linux Kernel is shown. The similarity of these tools is noticeable: both organize the available features in a tree-oriented style, the product configuration is performed by the individual selection of each required feature, and finally, both of them generate a file with the complete specification that is used for product generation.

3.3. Product Generation

The configuration and product generation implementation in the Linux Kernel is based on a set of small scripts called **kbuild**.

Firstly, kbuild implements the meta model for the configuration tree, which allows defining a hierarchy of configuration options; additional constraints specify the boundaries for valid configurations (see Figure 2 for an example). In terms of expressiveness, the used meta model is comparable to Czarnecki's definition of *feature models* [5]: Hierarchical composition, feature inter-dependencies, and additional constraints.

Secondly, kbuild is responsible for generating a Linux Kernel variant from a given configuration. The aforementioned configuration user interface creates a file (".config") holding a list of options the user selected; based on this configuration information kbuild generates a C header file with macro definitions in order to control inclusion or exclusion of code in the final product. Makefile build scripts finally control the compilation and linking of the resulting kernel image.

[4] from a trivial research operating system product line

3.4. Development Process

The reference process for SPL engineering [3, 8, 9] comprises iterative steps of refining analysis and design phases that allow industrial SPL manufacturers to keep the development controllable and efficient. The outcomes of these activities are *explicit* definitions of commonalities and variation points of the SPL on the requirements, software architecture, and implementation artefact levels.

In contrast, the Linux Kernel development process does not employ a uniform domain engineering process. Commonalities and variation points often emerge *implicitly* from implementation necessities and are not subject to a beforehand planning process. Even more profane Kernel evolution issues like new features or software architectural changes do not undergo a controlled planning process.

This latent lack of planning gets compensated by very large manpower: several thousands of volunteers (alongside with a few dozens of full-time paid engineers in contributing companies) stand by to implement new features, to peer review code changes, to do kernel-wide interface refactorings, or to support the release process by beta testing the kernel on their machines [7, 13].

Opposed to industrial SPL manufacturers, the Linux Kernel developers can afford this luxury – there do not exist any real release deadlines (although Linus Torvalds established a release cycle for the 2.6 kernel series, a release is still "done when it's done" [13]) or paying customers that need to be satisfied.

```
config PARAVIRT
   bool "Paravirtualization support (EXPERIMENTAL)"
   depends on EXPERIMENTAL
   depends on !(X86_VISWS || X86_VOYAGER)
   help
     Paravirtualization is a way of running multiple
     instances of Linux on the same machine, under a
     hypervisor. This option changes the kernel so it
     can modify itself when it is run under a
     hypervisor, improving performance significantly.
     However, when run without a hypervisor the kernel
     is theoretically slower. If in doubt, say N.

config VMI
   bool "VMI Paravirt-ops support"
   depends on PARAVIRT
   help
     VMI provides a paravirtualized interface to the
     VMware ESX server (it could be used by other
     hypervisors in theory too, but is not at the
     moment), by linking the kernel to a GPL-ed ROM
     module provided by the hypervisor.
```

Figure 2. Definition of configuration options with dependencies in the Linux Kernel

4. Discussion

In order to achieve an effective development of SPLs, the scientific community has provided guidelines like product line scoping, domain engineering, feature modeling, core asset development, etc. The Linux Kernel does not use any of these approaches (as described by the SPL community) in its development, and therefore cannot be considered an SPL.

The SPL guidelines aim at goals like automatic product generation, flexible architecture, high configurability, code reuse, no overhead of unrequested features, etc. The great success of the Linux Kernel is due to its ability to accomplish most of these goals, therefore, from this perspective, one could consider it to be an SPL. Additionally, the techniques used for kernel configuration and generation are very similar to the ones applied in the context of SPLs for product configuration and generation. Furthermore, in our opinion, the Linux Kernel (as well as SPLs in general) differs from simply highly configurable software as it provides a platform which encompasses multiple products being developed simultaneously.

These assumptions could raise the question whether all currently advertised SPL practices are necessary for building a competitive SPL. We think, they are absolutely necessary, as we believe the success of the Linux Kernel is very much related to its development process. The Linux Kernel has a very peculiar development process, which cannot be applied (at least in the proportions of the Linux Kernel) by a regular software company. Therefore, the SPL guidelines are of great importance to ease the development of SPLs in environments where limited resources (e.g. manpower), strict schedules and deadlines, and business interests must also be taken into consideration.

5. Conclusion

This position paper demonstrates how the Linux Kernel achieves some of the goals that the SPL guidelines also aim at. We have shown that a powerful development process without any rigorous software engineering procedures may achieve similar results as the ones expected from the practices offered by the scientific community.

However, we also concluded that the formalities for the SPL development are of crucial importance in more constrained development scenarios, so that effective and practical development practices can reduce the development efforts.

References

[1] J. Bermejo and N. Dai. Open source strengths for defining software product line practices. In *1st International Workshop on Open Source Software and Product Lines (SPLC 2006)*, 2006.

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

[3] G. Böckle, P. Knauber, K. Pohl, and K. Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag GmbH, Heidelberg, 2004.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[6] J. Greenfield and K. Short. *Software Factories*. Wiley, 2004.

[7] A. Morton. The linux kernel development process. http://aycinena.com/index2/index3/archive/andrew%20morton%20&%20the%20linux%20kernel.html, 2005.

[8] L. Northrop and P. Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[9] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[10] H. Schirmeier and O. Spinczyk. Tailoring infrastructure software product lines by static application analysis. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, 2007. (to appear).

[11] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat. On the Configuration of Non-Functional Properties in Software Product Lines. In *Proceedings of the 11th Software Product Line Conference, Doctoral Symposium (SPLC '07)*, 2007.

[12] M. Svahnberg and J. Bosch. Issues concerning variability in software product lines. In *IW-SAPF*, pages 146–157, 2000.

[13] J. van Gurp. OSS Product Family Engineering. In *1st International Workshop on Open Source Software and Product Lines (SPLC 2006)*, 2006.

[14] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.