A Glimpse on Generative Programming from the Operating Systems Perspective

Markus Pizka — position paper

Institut für Informatik, Technische Universität München, Germany - 80290 Munich
pizka@in.tum.de, http://www.in.tum.de/~pizka

Abstract

Effective generative programming techniques are of outstanding importance. Soon, the ability to employ generative techniques will determine both the rate of innovation and competition on future IT markets. This challenge must be addresed in general, independent from any domain of application.

In this position paper it is argued that operating systems (OS) are an interesting field to study aspects of general purpose generative programming (GP). An OS essentially generates implementations from abstract incomplete specifications. This task is fully automated and takes both, functional and non-functional requirements into account.

1 GP and IT Markets

Recently, Oracle CEO L. Ellison caused a stir by stating that the development of information technology (IT) was nearly completed.

In fact, the implementation of new innovative ideas often fails due to economical constraints. Consequently, new ideas are often abandoned a priori because of the awareness of economic restrictions. Thus, the innovation cycle becomes dull. Furthermore, the labour intense task of computer programming favors financially strong companies. First, only these companies are able to afford the development of new technologies. And second, big companies possess enough human power to rapidly provide an even more mature product if a small company tries to enter a market with a new idea.

Both aspects, the slackness of innovation and imperfect competition are obstacles for further rapid advances in IT. And both are mainly due to the tremendous costs for hand-coding software. Thus, generative programming (GP) [B+85, Sim95, Goe99, CE00] techniques are of predominant importance.

2 The OS Analogy

Operating systems (OS) are an excellent example for the above stated difficulties.The failure of IBM's attempt to disseminate OS\2 with superior concepts is just one prominent example for many other similar experiments like Mach [ABG+86] and BeOS [The97]. The main obstacle for innovations at the OS level is the constraint to stay compatible. OS developers tempt to avoid this conflict by adding layers, often called *middleware* or *runtime system*, on top of existing systems instead of performing a sound adaption of an integrated OS architecture. Examples such as CORBA [YD96] and POSIX Threading [IEE95] illustrate this phenomenon and its often negative consequences — e. g. multi-threading is unsupported by the memory management subsystem [Piz99b].

2.1 OS Needs

Clearly, the widespread use of GP techniques would significantly facilitate this difficult situation. GP would allow to regenerate new representations of application level software from abstract specifications according to changes made to the underlying OS.

The success of Linux and OpenSource is partially based on this phenomenon. The OpenSource model allows to rapidly adapt application level software to chances made to the underlying layers. Software packages are coded in a generic style and tailored to the execution environment during compile-time with configuration and system header files. Unfortunately, this generative power is all but systematically exploited but only used in specific cases, such as distinguishing big endians from little ones.

2.2 OS Provisions

Apart from the potential benefits of GP techniques in the OS field, OS are vice versa an interesting field to study generative techniques. The main two reasons for this proposition are:

1. Operating systems[1] are powerful general purpose generators. An OS automates a dynamic transition from abstract and incomplete requirements to a concrete, technical representation via a large number of intermediate refinements.

2. During this transformation, the OS often copes with functional as well as with large-scale non-functional and even contradicting requirements, such as timing constraints, security and reliability.

2.2.1 OS are Powerful Generators

Claim 1 is based on the observation that the task of an OS can be viewed as mapping abstract requirements given by a specification $w \in L$ using a high level programming language L (e. g. Java, Pascal, etc.) and transforming this specification to an element $h \in H$ of the hardware language H; $OS : L \rightarrow H$. Each specification w represents an application to be executed. w is typically incomplete requiring additional input during evaluation (otherwise w would be trivial, i. e. a constant). If input is needed, the operating system suspends execution and waits for input from devices or the execution of another specification w' before continuing with the evaluation of w.

Alternative refinements of the specification w with varying qualitative and quantitative aspects are reflected in simultaneously provided transition targets. Based on information I, thresholds T and criteria C the OS dynamically decides for a target resource R (i. e. an implementation variant) $I \times T \times C \rightarrow R$. This concept is intensively used on different levels of abstractions. E. g. depending on the load situation, an application level thread might either mapped to a user level or a kernel level thread.

[1] with the term "operating system" we subsume all kind of system level software including compiler, linker, runtime libraries and the kernel

A good example illustrating the incremental and dynamic nature of the automated generation of the result $h = OS(w)$ is dynamic paging. The set of program variables is partitioned into frames, stacks and statically allocated portions. Fragments of these partitions are dynamically transfered between main and secondary memory. Thus, the OS dynamically generates differently refined implementations of the abstract concept "program variable" depending on the availability of physical resources. In other words, the OS generates elements of a production line using a domain language hierarchy for storage management problems (e. g. activation frame, stack, segment, page).

2.2.2 Treatment of Mixed Requirements

Concerning claim, 2 it is well-known that OS not only transform functional requirements into a semantically equivalent representation at the hardware level but also optimize the output according to qualitative and quantitative requirements. E. g. a user might ask for high processing speed and at the same time for protection of his data. The ability of OS to cope with such requirements is remarkable from the GP point of view. It exposes similarities with the treatment of orthogonal requirements in aspect oriented programming [Cor00]. OS allow to study the effects of additional requirements besides correctness on the architecture, the design and the operation of generators.

2.2.3 Deficiencies

Unfortunately, these generative aspects of OS are neither developed systematically nor well understood. The hierarchy of domain specific languages, such as the storage system language hierarchy, exists solely implicitly. It's definition is scattered throughout function interfaces, macros and predominantly hand-coded into the behavior of the OS.

Nevertheless, OS are an interesting field for studying the interdependencies between domain specific languages, automated generation via several refinement steps, semi-automated transformation with user input and generation strategies in the presence of non-functional requirements.

3 Transfer of Knowledge

This rudimentary and non-formal discussion should make it clear, that generative programming could strongly benefit from OS experiences. Vice versa, simple GP techniques are already successfully used in the context of OS to generate production lines of OS with specific properties. Linux with its complex kernel configuration and recompilation facility including modules with deferred compilation can be regarded as a first step towards the successful generation of OS production lines. Although the high level language — the set of possible configurations — is still limited, this approach already leverages significant advantages.

The further exploitation of these possibilities requires a a sound conceptual funding of GP techniques. We propose an approach to develop a GP infrastructure based on experiences collected with the top-down development of a language-based distributed OS [EP99, Piz99a]. Instead of using a single wide-spectrum language or reversely restricting generality to a certain domain the approach distinguishes levels of abstractions of the specification language. Each transition from a level of abstraction i to a lower level $i+1$ refines the specification w_i given on level i to w_{i+1} on level $i+1$ as usual. Simultaneously, the grammar G of the specification language itself is gradually refined to reflect the needs of changed expressiveness.

$$G_i(N_i, T_i, P_i, S_i) \rightarrow G_{i+1}(N_{i+1}, T_{i+1}, P_{i+1}, S_{i+1})$$

For example, at a high level of the transition process we need to express abstract program variables whereas on a low level we must be able to specify a certain hardware register.

We state that this approach delivers new insights on the transition process performed by an OS. Furthermore, we believe that a thorough distinction between regular, context-free, context-sensitive and unrestricted parts of the specification language system induced by G_i is a promising starting point for the systematic automation of the generation process.

References

[ABG+86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[B+85] F. L. Bauer et al. The munich project CIP, vol. 1: The wide spectrum language CIP-L. volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1985.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

[Cor00] XEROX Corporation. Aspect-oriented programming home page. http://www.parc.xerox.com/aop/, 2000.

[EP99] C. Eckert and M. Pizka. Improving resource management in distributed systems using language-level structuring concepts. *The Journal of Supercomputing*, 13(1):33–55, January 1999.

[Goe99] Wolfgang Goebl. A survey and a categorization scheme of automatic programming systems. In *Lecture Notes in Artificial Intelligence*, volume 1799 of *LNCS*, page 1 ff., Heidelberg, 1999. Springer Verlag.

[IEE95] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[Piz99a] Markus Pizka. *Integriertes Management erweiterbarer verteilter Systeme*. PhD thesis, Technische Universität München, 1999.

[Piz99b] Markus Pizka. Thread segment stacks. In *Proc. of PDPTA '99*, Las Vegas, NV, June 1999.

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[The97] The Be Development Team. *The Be Developer's Guide: The Official Documentation for the BeOS*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1997.

[YD96] Z. Yang and K. Duddy. Corba: A platform for distributed object computing. *Operating Systems Review*, 30(2):4 – 31, April 1996.