An Introduction to GenVoca: Selected Papers (1992-1996)

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

1 Introduction

Building customized software systems from components has long been the dream of software engineers. The success of hardware Plug-and-Play — the ability to easily assemble off-the-shelf hardware components into customized configurations — presently has no counterpart in software. The most relevant research today is at the intersection of the areas of software reuse and software architectures.

The goal of *software architectures*, broadly speaking, is to identify techniques, solutions, tools, and tradeoffs that will make Plug-and-Play software a reality. *Software reuse*, or the reuse of components, is vital to architectures. *Software system generators* overlaps both areas. The distinguishing feature of generators is that its components are designed to be plug-compatible, interchangeable, and interoperable. Generator components are coded to standards that significantly simplify the problems of component reuse, customization, composition, optimization, and validation of compositions.

McIlroy outlined some of the key issues of Plug-and-Play software in a visionary paper in 1968 [McI68]; only in the last few years has noticeable engineering progress been made in making this vision a reality. We now know that there are many reasons why Plug-and-Play software has been slow to develop; the simplest reason is that the problem is much more difficult than any one imagined. The history of my involvement with generators underscores the challenges.

2 Why Generator Technologies are Slow to Develop

1977-1982. In a previous academic lifetime, I was a database researcher. My interests were in analytic models of database system (DBMS) performance. As a Ph.D. student at Toronto (1977-1980), it became clear to me upon an exhaustive reading of the literature that researchers used a small set of core ideas, assumptions, and techniques in hand-crafting performance models for customized database applications. Although I didn't know it at the time, the result of my dissertation was a generator for a domain of performance models: specific combinations and instantiations of its basic concepts reproduced the analytic equations and results of other researchers. The model, called the *Unifying Model* [Bat82a], not only related previously disparate results, it also presented a framework in which existing results could be generalized [Bat82b].

1983-1985. The Unifying Model was theoretical; it had never been applied to a real DBMS. Naturally, I was curious to know how practical it was. Upon collecting and studying voluminous man-

uals and notes (sometimes containing proprietary information) on how over ten different DBMSs were implemented, it was clear to me that the Unifying Model (and by implication, the models that it subsumed) was not general enough to describe commercial DBMSs accurately. A fundamentally more general approach was needed.

The central problem was that there was a vast number of ways of implementing database relations; the Unifying Model captured only the most elementary methods. To explain the storage structures used in commercial DBMSs required elementary structures to be composed. Here is where I discovered that the *storage architectures* (i.e., file structures, search and update algorithms) of a broad family of DBMSs could be explained as compositions of a small number of highly parameterized program *refinements*. With this new approach, called the *Transformation Model* [Bat85], I was able to capture and explain in precise detail — yet specify in a very compact way — how different DBMSs stored and retrieved data.

1985-1989. The next step was to validate the Transformation Model; its refinements had straightforward implementations as layers of hierarchical software systems. This took many years of work, but led to the Genesis DBMS generator, the first Plug-and-Play software technology for DBMSs [Bat88a-b]. Although parts of Genesis were operational in 1987, the first public/conference demonstration of Genesis was in 1989.

1990-1992. Long before Genesis was completed, it was evident to me that the techniques that I used to decompose the domain of relational DBMSs into components had nothing to do with relational DBMSs per se. It was not until the Fall of 1990 that I became familiar with Dr. Sean O'Malley and his work on Avoca/*x*-kernel [Hut89]. Avoca was the twin of Genesis in the domain of network protocols. By comparing and contrasting the designs of our two systems, the basic principles of decomposing domains into realms of plug-compatible components became evident. Our joint work lead to the first paper on GenVoca (a name synthesized from *Gen*esis and A*voca*) [Bat92].

1992-1996. In the following years, there were several important directions that I explored simultaneously. First was the issue of *generality*: could GenVoca be used to develop generators for other domains? Second was *performance*: could high-performance software systems be assembled from components? Third was *basics*: are the concepts of GenVoca fundamental (and not a complicated rehash of simple ideas)? If "no" was the answer to any question, GenVoca would likely be of interest only to academics. In the paragraphs below, I elaborate the progress made on each topic.

Generality. An essential ingredient in creating domain models is domain expertise. By 1992, I had exhausted the list of domains that I had enough expertise to create a domain model. Thus, I was fortunate to participate in the ADAGE project as the "domain modeling" expert to help design a generator for avionics software (i.e., data source, navigation, guidance, and flight director software). I interviewed avionics experts to define the ADAGE domain model; this model served as the blue-print for the ADAGE generator [Bat95a]. Furthermore, ADAGE's graphical editor (that allowed avionics software designers to specify a target system as a composition of components) was based largely on the graphical layout editor developed for Genesis.

At the same time I worked on ADAGE, I discovered yet another independently-conceived generator, Ficus [Hei93], that fit the GenVoca paradigm. Both ADAGE and Ficus provided critical evidence on the generality of GenVoca.

Performance. Genesis demonstrated the feasibility of assembling customized DBMSs from components, but it did not convincingly show that the DBMSs produced were efficient. Genesis made me aware of the tremendous opportunities for optimization if a generative approach, rather than compositional approach, were taken. Genesis was a *compositional* generator: its components were composed and executed at DBMS run-time. This meant that all domain-specific optimizations that could simplify a target DBMS and substantially increase its performance had to be applied and executed at DBMS run-time with a concomitant enormous overhead. In contrast, generative components output code that is to be executed by applications. Domain-specific optimizations and simplifications could be applied at DBMS/application generation-time, *not* at DBMS/application run-time. Consequently, generative components could potentially produce very efficient software.

Although it was not initially obvious, this line of reasoning exposed a fundamental connection of GenVoca to program transformation systems. GenVoca refinements can be implemented as modules (as Genesis and Avoca demonstrated), but they could also be implemented as transforms of a program transformation system. Coding GenVoca refinements as transforms was a simple way to introduce and apply domain-specific optimizations at software-generation time.

The P2 generator for container data structures was the first GenVoca generator implemented as a program transformation system. (P2 components mapped parse trees of abstract programs into parse trees of less abstract (or concrete) programs). Although we had many "small" successes in generating efficient code, our most ambitious undertaking and most significant results came when we used P2 to re-engineer LEAPS. LEAPS is a performance-driven, hand-coded, highly-tuned production system compiler [Mir90]. At the time of our work, LEAPS produced the fastest executables of OPS5 rule sets. It converted OPS5 rule sets into C programs that utilized unusual container data structures and search algorithms for efficient rule processing. It was well-known that the complexity of LEAPS made it difficult to understand and build. Although my students and I had no prior experience with production systems, we were able to rebuild LEAPS using P2 in less time than experts (who had built LEAPS-like systems before). Moreover, P2 reduced the volume of programming by a factor of four; the LEAPS algorithms had a clean, elegant, and easy-to-understand specification as P2 programs; and the performance of our generated C programs was better than that of LEAPS. (P2 could perform complex domain-specific optimizations automatically that were difficult, if not impractical, to do by hand). Moreover, the simplicity of our P2 specifications of the LEAPS algorithms suggested a way to improve performance dramatically. By altering the composition of components that implemented LEAPS containers, we were able to improve performance by *orders* of magnitude in the execution of some rule sets [Bat94, Bat95b]. These results, coupled with results from Avoca/*x*-kernel and Ficus projects, convinced us that efficient software could be produced by generators.

Basics. GenVoca is an unorthodox design and programming paradigm. Recognizing the concepts to explain its underpinnings has always been a challenge. Some concepts were evident early on: layers and virtual machines, large-scale program transformations (i.e., refinements that simultaneously and consistently refine multiple classes of a software design), parameterized

programming (a simple and elegant model of software composition), and symmetry (components exporting and importing the same interface). However, new concerns have since arisen that have an elegant expression in GenVoca, suggesting that there are indeed basic principles at work.

An important issue in generators is determining whether a composition of components is *consistent*. There must be automated help in validating and debugging compositions, otherwise users might be clueless on how to repair their component assemblies. GenVoca domain models are attribute grammars: tokens correspond to components and sentences correspond to systems (component compositions). It is well-known that not all syntactically correct programs are semantically correct; the same idea holds for compositions of components. Attributes encode the conditions in which components can be used legally. This insight led to the development of very simple and efficient algorithms for validating component compositions [Bat96a].

A second issue is *scalability*: a small number of components should be composable in vast numbers of ways to produce large families of software systems. The connection of GenVoca domain models to grammars demonstrates scalability: adding a new token to a grammar is equivalent to adding a new component to a GenVoca library. The number of new sentences (systems) that can be produced (composed) with a new token (component) is potentially enormous [Bat92-93].

A third issue stems from a phenomenon called *subjectivity* [Har93]: when modeling families of applications, objects do not have single interfaces, but are described by a family of related interfaces. The interface that is appropriate for an object will be application-dependent. Because GenVoca models families of systems, the effects of subjectivity must be accounted for. Although the simplest explanation of GenVoca assumes components that export and import standardized interfaces, the true story is more complicated. It turns out that GenVoca components customize themselves upon instantiation: their interfaces and bodies automatically adjust to the requirements of the system in which they are being used. This behavior is not yet well-understood, but has been observed in all GenVoca generators. The best explanation of subjectivity to date suggests that the ideas of operation propagation (via subclassing) and method wrappers (of meta-object protocols) have counterparts in parameterized programming [Bat96b].

What's Next? Basic research is still needed. The development of GenVoca has been, and will continue to be, experimentally-driven. Many of its distinguishing (and nonobvious) features arose out of practical necessities in making generators solve real-world problems. The more domains that are analyzed, the better our understanding will be about the principles at work.

Constructing GenVoca generators has been very labor-intensive. All of the above-mentioned generators have been built from scratch because there is no tool support for GenVoca. Over 80% of the construction effort in building the infrastructure of GenVoca generators is writing languages (and their compilers) to define components, developing tools to compose components, and providing other domain-independent environmental support (e.g., component composition consistency checkers, layout editors). Not surprisingly, the high start-up costs of building generators discourages attempts at creating them. To move generator technology forward will require an advance in programming languages; it is not cost effective to write generators from scratch.

Recap. So why have generator technologies been so difficult to develop? From my experience, generators rely on an unorthodox programming technology that takes a long time to understand. It requires domain expertise and insight to know how to decompose domains into reusable components. It takes time and effort to validate models by building generators and to re-engineer nontrivial applications to evaluate the potential of generators. To understand the domain-independent principles of generator organization requires building generators for different domains. Lastly, it is a nontrivial intellectual challenge to accomplish all this because one needs expertise in many areas of software design and development (e.g., compilers, transformation systems, parameterized programming, object-oriented programming, meta-object protocols, and software architectures).

3 This Volume of Papers

The future of software engineering lies in automation: programmer productivity can be greatly increased by programming at higher-levels of abstraction and producing rote software automatically. I know that generators can realize these lofty goals, and my research is to show how this can be accomplished.

To help others avoid the pitfalls and the innumerable dead-ends that I encountered over the years, I have created this collection of papers that give my best explanation to date of GenVoca and its potential. The first paper is my ACM TOSEM paper with O'Malley [Bat92]. The second paper, "Scalable Software Libraries", explains why conventional ways of building software libraries are inherently limited and why scalability is vital to domain-specific libraries [Bat93]. The third paper, "Creating Reference Architectures..." shows the power of GenVoca concepts and outlines the blueprint (domain model) for the ADAGE avionics software generator [Bat95a]. (This paper was voted the Best Paper at the *1995 Symposium for Software Reuse*). The fourth and fifth papers, "Validating Component Compositions..." [Bat96a] and "Subjectivity..." [Bat96b], were two of four papers recommended for journal publication from the *1996 International Conference on Software Reuse*. Finally, the last two papers present experimental results on LEAPS that show performance and productivity improvements that generators can deliver [Bat94, Bat95b].

4 Table of Contents

5 References

[Bat82a] D.S. Batory and C.C. Gotlieb, “A Unifying Model of Physical Databases”, *ACM Transactions on Database Systems*, 7,4 (December 1982), 509-539.

[Bat82b] D.S. Batory, “Optimal File Designs and Reorganization Points”, *ACM Transactions on Database Systems*, 7,1 (March 1982), 463-528.

[Bat85] D.S. Batory, “Modeling the Storage Architectures of Commercial Database Systems”, *ACM Transactions on Database Systems*, 10,4 (December 1985), 463-528.

[Bat88a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C.Twichell, T.E. Wise, “GENESIS: An Extensible Database Management System”, *IEEE Transactions on Software Engineering*, 14,11 (November 1988), 1711-1730.

[Bat88b] D.S. Batory, “Concepts for a Database System Synthesizer”, *ACM Principles Of Database Systems Conference 1988*, 184-192.

[Bat92] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT* 1993.

[Bat94] D. Batory, J. Thomas, and M. Sirkin, “Re-engineering a Complex Application Using a Scalable Data Structure Compiler”, *ACM SIGSOFT* 1994.

[Bat95a] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, “Creating Reference Architectures: An Example From Avionics”, *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.

[Bat95b] D. Batory and J. Thomas. “P2: A Lightweight DBMS Generator”, Technical Report TR-95-26, Department of Computer Sciences, University of Texas at Austin, June 1995.

[Bat96a] D. Batory and B.J. Geraci, “Validating Component Compositions in Software System Generators”, *Fourth International Conference on Software Reuse*, Orlando, Florida, April 1996.

[Bat96b] D. Batory, “Subjectivity and GenVoca Generators”, *Fourth International Conference on Software Reuse*, Orlando, Florida, April 1996.

[McI68] D. McIlroy, “Mass Produced Software Components”, Software Engineering: Report on a Conference by the Nato Science Committee, Oct 1968, P. Naur and B. Randell, eds. 138-150.

[Mir90] D. Miranker, D. Brant, B. Lofaso, and D. Gadbois, “On the Performance of Lazy Matching in Production Systems”, *Proceedings of the National Conference on Artificial Intelligence*, 1990.

[Har93] W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, *OOPSLA 1993*, 411-428.

[Hei93] J.S. Heidemann and G.J. Popek, “File-System Development with Stackable Layers”, *ACM Transactions on Computer Systems*, 12(1), 58-89.

[Hut89] N. Hutchinson, L. Peterson, S. O’Malley, and M. Abbott, “RPC in the *x*-kernel: Evaluating New Design Technique”, *Symposium on Operating System Principles*, (December 1989), 91-101.