

Software Generators, Architectures, and Reuse

Don Batory

**Department of Computer Sciences
University of Texas
Austin, Texas 78712**

**batory@cs.utexas.edu
512-471-9713
<http://www.cs.utexas.edu/users/dsb/>**

Introduction

Tutorial answers following questions:

- what is a (reusable) building block or *component* of a domain?
- what are guidelines for creating software component technologies?
- how can complex software systems be generated or synthesized from components?
- how are software architectures and generators related?

Will review answers that have been distilled from actual experiences and prototype software generators:

- for well-understood domains of hierarchical (layered) software systems
- take engineering look at fundamental concepts that underlie actual generators

So What?

Software Reuse

- generators are success stories; show secrets of success

Software Architectures

- generating family of systems requires definition of architecture; generators are exemplars to study

Domain Modeling / Software Modeling

- how do you model a family of systems?
rather unusual modeling/programming paradigm

Programming Languages

- seeking concepts, linguistic features to encapsulate components, express generator concepts

Compilers

- generators are compilers; assembly languages are Java, C, or C++; many domain-specific optimizations

Industrial Usage

- ideas of tutorial are being applied in industry

Organization of Tutorial

Sequence of short lectures (with question/answer periods):

Lecture / Period		Length
Optional Lectures (Choose 1)	Introduction to GenVoca	50 min
	An Example	55 min
	Break	
	Subjectivity	40 min
	Domain Modeling Methodology	40 min
	Design Rule Checking	40 min
	Architectural Styles	40 min
Recap & Open Discussion		

**Qualification: rich subject area that
requires familiarity with many topics!**

State of Software System Development

Developing large systems: difficult, time consuming, high cost, high risk

- future trends not encouraging:

complexity	increasing
delivery time	increasing
scalability	build from scratch
reinvention	massive
evolution	living fossils

Next generation environments based on domain-specific large scale reuse technologies

reduces	complexity
	delivery time
	reinvention
scalable	add new components
evolution	generator technology

Problems: What is large scale reuse?
How to achieve large scale reuse?

Overview of Reuse Granularities

SSR	small scale reuse	algorithm, function reuse
MSR	medium scale reuse	suites of related functions (classes)
LSR	large scale reuse	suites of related classes (subsystems, frameworks)

No general theory, proven principles for LSR

- few examples of LSR technology

Genesis	Batory: University of Texas
1988	customize DBMSs by composing prefabricated components
	goal: demonstrate feasibility of building-block approach to DBMS construction
Avoca/x-kernel	O'Malley & Peterson: University of Arizona
1989	customize protocol suites by composing prefabricated protocols
	goal: show that highly layered protocols can be efficient

- when we compared notes...

Tutorial Overview

Both projects were virtually identical in conception and implementation

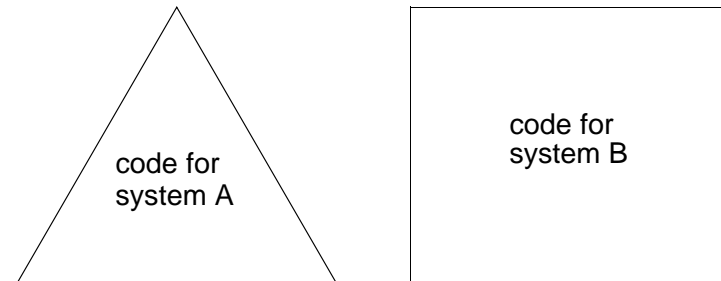
- similarities not accidental
- Genesis & Avoca/x-kernel are instances of domain-independent theory of large scale reuse and software generation
- tutorial reviews general “theory” & principles

“GenVoca” name given to common ideas

GenVoca exposes fundamental role of domain modeling in large scale reuse and system generation

- what is domain modeling (reference architectures)?
- what is relationship to large scale reuse?

Domain Modeling



Rich Set of Lessons Learned

Ad Hoc Software Designs/Decompositions

- don't work for generators and large scale reuse
- consequence of conventional 1-of-kind system designs
- not suitable for assembling families of systems

Large Scale Software Reuse

- is a consequence of premeditated design;
standardization of recurring “shapes” within a domain

programming abstractions and implementations are standardized

- components must be designed to be interoperable and composable

components will not have these properties otherwise

Lessons Learned (Continued)

Domain Modeling

- is retrospective study of a family of systems
- differs from application modeling (i.e. point designs)
- process of standardizing well-understood domain
- parametric model of software systems
- blueprint for domain-specific software generator

Software System Generators

- can significantly increase productivity
- especially if all required components are available

Explain GenVoca from first principles

- goes beyond OO concepts
- OO is medium scale, not large scale

Resist Interpretation!!
Look at syntax now, semantics later!!

Background on Hierarchical Software

Virtual machines (Dijkstra 1968)

- design each level of a hierarchical system independently
- each level defines a *virtual machine*

all operations on level $i+1$ defined in terms of operations on level i

Refresh using OO ideas:

- define interface as objects + operations
- hierarchical design is set of:

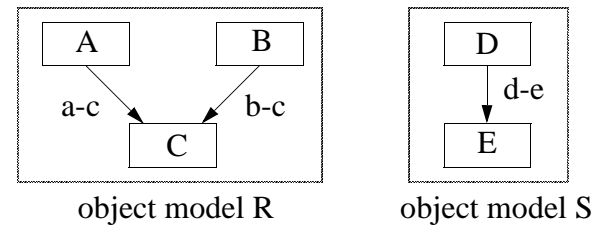
object oriented virtual machines (OOVM)

one OOVM for each level

Background

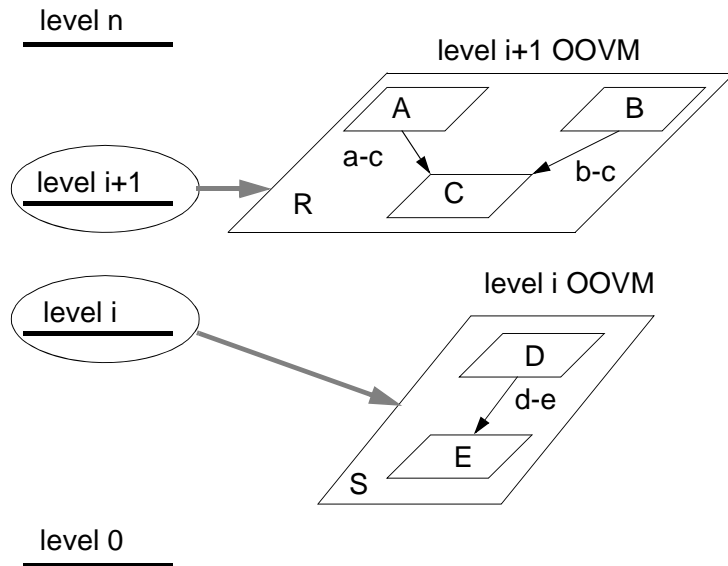
Object model (or *object-oriented virtual machine*) is set of classes and their interrelationships

Use ER notation:



GenVoca not dependent on specific variant of ER

Hierarchical Design



Layer/component is:

- a **refinement** (*mapping* or *transformation*) between virtual machines
- **large scale** — simultaneous and consistent refinement of multiple classes, objects, and methods

What is New?

Large scale components are **layers**

- are encapsulated suites of interrelated classes
- are new: larger units of abstraction and encapsulation (see lecture on **Subjectivity**)

How is large scale reuse achieved?

- standardize fundamental abstractions, OOVMs of a well-understood domain
- *layers export & import standardized interfaces*
- *layers are designed to be plug-compatible, interchangeable, interoperable*

latter two points are trademarks of GenVoca designs

- new: contrary to traditional library/reuse paradigms

More difficult to achieve:

- substantial step beyond design of OOVMs
- key to software generation

The GenVoca Model

Component (layer) is fundamental unit of large scale software construction

- interface of component is an object model

(multiple classes, relationships)

- component **w** exports OOVM interface **S**

All components that export the same interface (OOVM) belong to a *realm*

- realm is a *library* of plug-compatible, interoperable, and interchangeable components

- OOVM **S** and **R** define realms **S** and **R**

$$S = \{ y, z, w \}$$

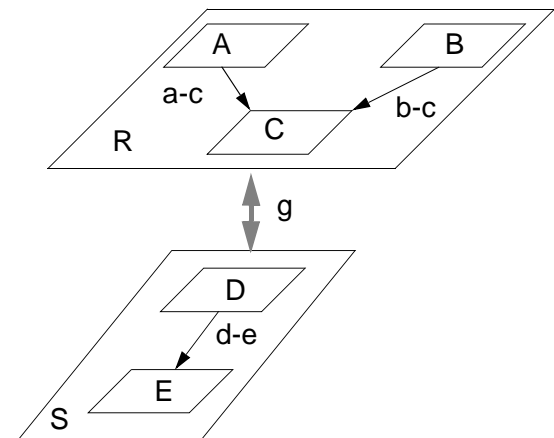
$$R = \{ g[x:S], h[x:S], i[x:S] \}$$

- note: components may be parameterized

Components with Parameters

Consider $g[x:S] : R$

- **g** exports interface **R**; **g** imports interface **S**



- **g** translates operations and objects of **R** to **S**
- *parameter $x:S$ means that translation doesn't depend on a specific implementation of **S***

Software Systems and Type Equations

A software system/subsystem is named composition of components called a *type equation*:

$$S = \{ y, z, w \}$$

$$R = \{ g[x:S], h[x:S], i[x:S] \}$$

$$\text{system1} = g[y];$$

$$\text{system2} = g[w];$$

$$\text{system3} = h[w];$$

*now possible to precisely
define family of systems
that can be built*

*can reason about systems
in terms of their components*

- modeling systems as equations is hallmark of *parameterized programming* (Goguen 1986)

Grammars and Families of Systems

Realms and components define a *grammar* whose sentences (component compositions) are software systems

Parameterized component representation:

$$S = \{ y, z, w \}$$

$$R = \{ g[x:S], h[x:S], i[x:S] \}$$

Grammar representation:

$$S := y \mid z \mid w$$

$$R := gS \mid hS \mid iS$$

The set of all sentences defines a language:

- Parnas family of systems (1976)
- connection with grammars goes further...

Symmetric Components

Just as recursion is fundamental to grammars, *symmetric components* are fundamental to GenVoca

- export and import same interface
- composable in virtually arbitrary orders

order of composition affects semantics & performance

- *symmetric* components of realm w have parameters of type w :

$$w = \{ m[x:w], n[x:w], p \}$$

$$w := m \ w \mid n \ w \mid p$$

- examples:

$$m[n[p]], n[m[p]], m[m[p]], n[n[p]]$$

- familiar example: Unix file filters

Scalability

Adding a new component to a realm is equivalent to adding a new rule to a grammar

- the family of systems enlarges exponentially (in length of type equation)
- because large families can be built using relatively few components, GenVoca models are *scalable*

Most contemporary software libraries, even STL

are *not* scalable...

are *not* suitable for software generators...

*See lecture on
An Example*

Component Reuse (LSR)

Obvious: different systems/equations reference the same component...

```
system1 = g[ y ];
```

```
system2 = g[ w ];
```

```
system3 = h[ w ];
```

- components **g** and **w** are reused...
- look for common subexpressions, common terms

Design Rules and Domain Models

Given realms below, in principle any component of **R** can be composed with any component of **S**:

$$\mathbf{S} = \{ \mathbf{y}, \mathbf{z}, \mathbf{w} \}$$

$$\mathbf{R} = \{ \mathbf{g}[\mathbf{x}:\mathbf{S}], \mathbf{h}[\mathbf{x}:\mathbf{S}], \mathbf{i}[\mathbf{x}:\mathbf{S}] \}$$

- although equations may be type correct, there are always combinations of components that don't make sense
- domain-specific constraints called *design rules* preclude illegal component combinations

GenVoca domain model is:

- realms of components
- design rules that restrict compositions
- can be expressed as an *attribute grammar*

*See lecture on
Design Rule Checking*

Generators and Architectures

A *generator* is an implementation of a domain model

- next lecture we will see that there are very different ways of implementing domain models

component parameters can be instantiated at run-time or at compile-time/generation-time

generators can have compositional, generative, transformational implementations...

An *architecture* is the structure of a system in terms of components

- a type equation defines the architecture of a system

*See lecture on
Architectural Styles*

Why GenVoca Important?

The simplest “building-blocks” model of software construction has components that import and export standardized interfaces

- idea that has arisen independently in domains

System	Domain	Year
Genesis	Database Management	1988
Avoca/x-kernel	Network Protocols	1989
Ficus	File Systems	1990
Rosetta	Database Data Languages	1994
ADAGE	Avionics	1994
ASP	Audio Signal Processing	1995
Jakarta	Extensible Precompilers	1997
P3	Data Structures	1997
LavaLamp	Radio Software	1997
...

- there common and deep problems of conceptualization and implementation that arise in every one of these domains/generators

This tutorial allows you to avoid costly reinvention...

Performance of Generated Software?

Good...

- most generators focussed on performance (not on proof-of-concept):

generated software comparable to that hand-written, hand-crafted by experts

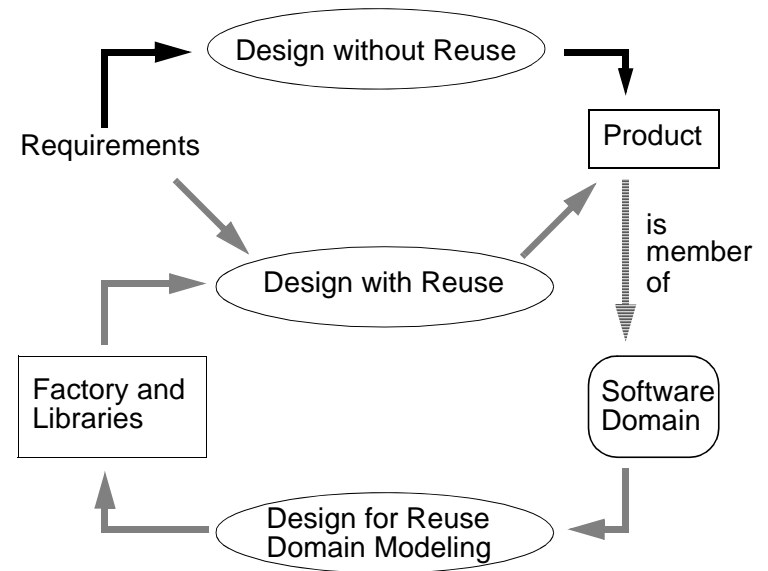
usually superior performance

(regenerate system using performance-enhancing algorithms)

substantial improvements in productivity (if most or all components available)

- see Reading List...

Software Factory Paradigm



Goal: faster, cheaper, better software development

GenVoca is one way to realize factory and libraries

Conclusions — Lessons Learned

Generating complex software systems from components has unusual requirements:

- ***subsystems/layers are fundamental building blocks***

multi-class encapsulations

layers export and import multi-class virtual machine interfaces

- ***standardizing abstractions, interfaces of domain to define realms (libraries) of plug-compatible components***

different than conventional library paradigms

- ***software systems are modeled by equations***

parameterized programming

Rest of tutorial will expose GenVoca in more detail

Further Reading

- D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.
- T. Biggerstaff, "An Assessment and Analysis of Software Reuse", in *Advances in Computers*, Volume 34, Academic Press, 1992.
- T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 102-110.
- E.W. Dijkstra, "The Structure of THE Multiprogramming System", *Communications of ACM*, May 1968, 341-346.
- J.A. Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer*, February 1986.
- R.E. Johnson and B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June/July 1988.
- C. Krueger, "Software Reuse", *ACM Computing Surveys*, June 1992, 131-183.
- 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.
- D.L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, March 1979.
- R. Prieto-Diaz and G. Arango (ed.), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press 1991.