Product-Line Architectures,

Aspects,

and Reuse

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

batory@cs.utexas.edu
512-471-9713
www.cs.utexas.edu/users/dsb/tutorial.html
My Background

In 1983, I began studying how database management systems could be synthesized from plug-compatible components. First journal paper on this subject appeared in ACM Transactions on Databases, December 1995

---

Central problems in software engineering stem from:

- easy to understand complaints about software quality, performance, reliability, maintainability, evolvability, …

Software quality is a function of:

- quality of application design
designs improve with experience
design is difficult — build several times to get it “right”

- quality of the programming staff

Answers to both are highly variable. So…

- how can we do better?
- how can we better exploit previous systems, experiences?
The Future (and this Tutorial)

The future of software development lies in standardization and automated production of well-understood software:
- major improvements in quality, reliability, performance…
- technology based on:

**Domain-Specific Component Technologies**

Principled engineering approach that standardizes:
- expert-approved designs (programming problems)
- expert-approved implementations (programming solutions)
- component compositions define target systems

In this way, we improve key factors of software quality:
- using “expert” designs, “expert” implementations

Key Features of Vision

Scale of “component” encapsulation/reuse is critical:

<table>
<thead>
<tr>
<th>Scale</th>
<th>Reuse Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSR</td>
<td>small scale reuse</td>
<td>algorithm, function reuse</td>
</tr>
<tr>
<td>MSR</td>
<td>medium scale reuse</td>
<td>suites of related functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(classes)</td>
</tr>
<tr>
<td>LSR</td>
<td>large scale reuse</td>
<td>suites of related classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(subsystems, frameworks)</td>
</tr>
</tbody>
</table>

Using LSR components is key; challenge is *how*?

Tutorial answers following questions:
- what is a large-scale component or building-block for a domain of applications?
- how are component-based product-line architectures defined?
- how can complex, efficient, and extensible applications be constructed from components?

We review answers distilled from experiences and prototype generators:
- tools that assemble families of systems from expert-designed & expert-implemented components for well-understood domains
So What?

Product-Line Architectures (PLAs)
• producing family of applications requires definition of a PLA; component-based generators are exemplars to study

Domain Modeling / Software Modeling
• how do you model a family of applications? state-of-art modeling/programming paradigm

Software Reuse
• component-based generators are success stories; reveal secrets of success

Relevance to Research
• Aspect Oriented Programming (AOP), Microsoft’s IP Intensions, Collaboration-Based Design, Perry’s Light Semantics, …

Relevance to Practice
• ideas of tutorial are being used in industry; Microsoft’s COM is a special case of this technology

Organization of Tutorial

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

<table>
<thead>
<tr>
<th>Lecture / Period</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to GenVoca</td>
<td>40 min</td>
</tr>
<tr>
<td>Mixin-Layer Implementations of Refinements</td>
<td>50 min</td>
</tr>
<tr>
<td>Design Rule Checking</td>
<td>40 min</td>
</tr>
<tr>
<td>Recap &amp; Open Discussion</td>
<td></td>
</tr>
</tbody>
</table>

Expanded Tutorial

<table>
<thead>
<tr>
<th></th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Modeling Methodology</td>
<td>40 min</td>
</tr>
<tr>
<td>Metaprogram Implementations of Refinements</td>
<td>40 min</td>
</tr>
<tr>
<td>Architectural Connectors as Refinements</td>
<td>30 min</td>
</tr>
<tr>
<td>Design Wizards and Automatic Selection of Components</td>
<td>30 min</td>
</tr>
</tbody>
</table>

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

Name given to a general theory and principles for software “legos”:

- ideas have surfaced (i.e., rediscovered, reimplemented) in many different contexts

- goal of tutorial is to avoid reinvention of these ideas

- name is a merging of the names “Genesis” and “Avoca” (the first two systems built on these principles)

idea of components that export and import standardized interfaces taken to its logical conclusion

GenVoca exposes fundamental role of domain modeling in large scale reuse, PLAs, and generation:

- what is domain modeling (reference architectures)?

- what is relationship to large scale reuse?
Rich Set of Lessons Learned

Ad Hoc Software Designs/Decompositions
• don’t work for component-based generators
• consequence of conventional 1-of-kind system designs
• not suitable for assembling product-lines economically

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 or blue-print of a PLA

Component-based 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, classes, and methods
• 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-like notation:

![Diagram](https://via.placeholder.com/150)

object model R

object model S

GenVoca not dependent on specific OO notation (choose your own …)
Hierarchical Design

- **Layer, component, “aspect”** is:
  - a **consistent 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 or aspects*
  - are encapsulated suites of interrelated classes or fragments of related classes
  - are new: larger units of abstraction and encapsulation (see next lecture on **Mixin-Layers**)

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

More difficult to achieve:

- new: contrary to traditional library/reuse paradigms
**The GenVoca Model**

*Component (layer, aspect)* 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`
  ```
  \begin{array}{c}
  \text{A} \\
  \text{a-c} \\
  \text{R} \\
  \text{b-c} \\
  \text{B} \\
  \text{g} \\
  \text{D} \\
  \text{d-e} \\
  \text{S} \\
  \text{E}
  \end{array}
  ```
- `g` translates operations and objects of `R` to `S`; translation called a *refinement*
- parameter `x:S` means that translation doesn’t depend on a specific implementation of `S`
Applications and Type Equations

An application/subsystem is named composition of components called a type equation:

\[
\begin{align*}
S &= \{ y, z, w \} \\
R &= \{ g[ x:S ], h[ x:S ], i[ x:S ] \}
\end{align*}
\]

app1 = g[ y ];
app2 = g[ w ];
app3 = h[ w ];

now possible to precisely define family of applications that can be built

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

Interpretation

Interpretation critical!
• relates domain features to components

\[
\begin{align*}
S &= \{ y, \quad \text{// program with feature } y \\
z, \quad \text{// program with feature } z \\
w \quad \text{// program with feature } w
\end{align*}
\]

\[
\begin{align*}
R &= \{ g[ x:S ], \quad \text{// adds feature } g \text{ to } x \\
h[ x:S ], \quad \text{// adds feature } h \text{ to } x \\
i[ x:S ] \quad \text{// adds feature } i \text{ to } x
\end{align*}
\]

So, type equations define programs with known features!

app1 = g[ y ]; // program w. features g,y
app2 = g[ w ]; // program w. features g,w
app3 = h[ w ]; // program w. features h,w

can reason about applications in terms of their components
Grammars and Families of Applications

Realms and components define a grammar whose sentences (component compositions) are applications.

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 (product-line):

- 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 \( W \):
  \[ W = \{ m[x:W], n[x:W], p \} \]
  \[ W := mW \mid nW \mid p \]
- examples:
  - \( m[n[p]], n[m[p]], m[m[p]], n[n[p]] \)
- familiar example: Unix file filters
Scalability and Component Reuse

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

- the family of applications enlarges exponentially (in length of type equation)

- because large families can be built using relatively few components, GenVoca models are scalable

Component reuse obvious: different systems/equations reference the same component...

```plaintext
application1 = g[y];
application2 = g[w];
application3 = h[w];
```

- components g and w are reused...
- reuse is 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:

```
S = { y, z, w }
```

```
R = { g[x:S], h[x:S], i[x: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
Important Special Case

Microsoft’s Component Object Model (COM)

- components can export and import “standardized” interfaces
- applications are compositions of COM components

Differences:
- supports multiple-inheritance among interfaces (like Java)
- allows components to export *multiple* standardized interfaces, and import components that implement *multiple* interfaces (R and S, R or S)
- very useful indeed, but not critical to our PLA model

COM can be used to create GenVoca product-lines, but that isn’t how COM is used today
- most interfaces implemented by a single component (IExplorer, Windows Media Player)
- different implementations arise from versioning
- generally don’t have the GenVoca plug-and-play

Special Case (Cont)

- Key restriction: COM interfaces limited to a *single class*

we will look at components that have more complicated (i.e., multi-class) interfaces. Our components are Java packages or Microsoft DLLs that are typed (e.g. have interfaces) — concept not present in today’s operating systems and programming languages

- Key restriction: COM components are binaries

binaries are NOT the only way refinements can be implemented: there are lots of other ways (see next few slides).

In fact, if one limits components only to (COM) binaries, there are many domains for which PLAs couldn’t be built — the assembled applications would run so slowly that no one would ever use them.

This doesn’t mean that PLAs can not be implemented for these domains, it simply means that refinements for this domain have to be implemented differently …
**Why GenVoca Important?**

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

- idea that has arisen independently many times

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genesis</td>
<td>Database Management</td>
<td>1988</td>
</tr>
<tr>
<td>Avoca/x-kernel</td>
<td>Network Protocols</td>
<td>1989</td>
</tr>
<tr>
<td>Ficus</td>
<td>File Systems</td>
<td>1990</td>
</tr>
<tr>
<td>Rosetta</td>
<td>Database Data Languages</td>
<td>1994</td>
</tr>
<tr>
<td>ADAGE</td>
<td>Avionics</td>
<td>1994</td>
</tr>
<tr>
<td>ASP</td>
<td>Audio Signal Processing</td>
<td>1995</td>
</tr>
<tr>
<td>ITS</td>
<td>Extensible Precompilers</td>
<td>1997</td>
</tr>
<tr>
<td>P3</td>
<td>Data Structures</td>
<td>1997</td>
</tr>
<tr>
<td>LavaLamp</td>
<td>Radio Software</td>
<td>1998</td>
</tr>
<tr>
<td>FSAT99</td>
<td>Command-and-Control Simulator</td>
<td>1999</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- there common and deep problems of conceptualization and implementation that arise in every one of these domains/generators and that take years to sort out…

This tutorial allows you to avoid costly reinvention…

**Perspective**

GenVoca advocates that the atomic building blocks of product-lines are **refinements** (i.e., mappings between standardized virtual machines)

- model doesn’t say *how* refinements are implemented or *when* refinements are composed
- all that is known is that refinements have parameters and can be composed via parameter instantiation

*Lack of specificity makes GenVoca general…*

**Refinements can have vastly different implementations:**

- **object**
  (Java object or COM component)
- **template**
  (mixin-layer)
- **metaprogram**
  (a program that generates another program)
- **rule set of a program transformation system**
  (transformations of abstract specifications into efficient programs through rewrites)
In addition to multiple implementations, refinements can be composed at different times:

- **statically** at application compile time (build time) (for applications whose type equation is fixed)
- **dynamically** at application execution time (allows type equations to change during program execution)

Look at known GenVoca PLAs:

<table>
<thead>
<tr>
<th>Refinement Implementation</th>
<th>Refinement Composition Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
</tr>
<tr>
<td>Object</td>
<td>Genesis</td>
</tr>
<tr>
<td></td>
<td>Ficus</td>
</tr>
<tr>
<td>Template</td>
<td>JTS</td>
</tr>
<tr>
<td></td>
<td>FSATS, ADAGE</td>
</tr>
<tr>
<td>MetaProgram</td>
<td>P3</td>
</tr>
<tr>
<td></td>
<td>LavaLamp</td>
</tr>
<tr>
<td>Transformation</td>
<td>?</td>
</tr>
</tbody>
</table>

Refinements can have different implementations and can be composed at different times, there is yet another variability — there are different kinds of refinements!

- **Virtual Machine Refinements**
  imported interface(s) are not visible consistent with standard notion of “virtual machines”

- **Subjective (Extension) Refinements**
  imported interfaces are visible enhancing lower-level VMs with more capabilities and exporting this enhanced VM — see mixin-layers

- **Optimizing Refinements**
  map less-efficient programs to more efficient programs

\[
\begin{align*}
i &= 3 + 4 - 5; \quad \Rightarrow \quad i = 2; \\
\text{sum} &= 0; \quad // \text{n} > 0 \quad \Rightarrow \quad j = n+1; \\
\text{for (j=1; j<=n; j++)} & \quad \text{sum} = \text{sum} + j; \\
\text{sum} &= (n\cdot j)/2;
\end{align*}
\]

note: subjective and optimizing refinements are symmetric
Perspective (Cont)

How to chose among implementations?
Ans: depends on application requirements

- dynamic for run-time reconfigurations
- static for optimizations
  although different technologies have different optimization possibilities:

  templates — none (except for C++)
  metaprograms — lots (but need language support)
  program transformation systems — ∞ (need rule engine)

GenVoca offers a single way in which to conceptualize a domain and its building blocks in a largely implementation-independent way

- ex: don’t use rule-sets when templates or objects suffice

Conceptual economy of GenVoca is a big win …

Conclusions — Lessons Learned

- GenVoca takes idea of components exporting and importing standardized interfaces to logical conclusion

- refinements are basic building blocks

  abstract design entities
  implementations are multi-class encapsulations
  layers export and import multi-class virtual machine interfaces

- standardizing abstractions, their interfaces and implementations as plug-compatible components

  different than conventional library paradigms

- applications are modeled by equations

  parameterized programming

- a variant of COM is an instance of GenVoca

Rest of tutorial will explain GenVoca in more detail
Further Reading


Mix-in Layer Implementations of Refinements

Understanding GenVoca requires looking at example domains where symmetry, scalability, encapsulation, and composition issues can be studied carefully.

In this lecture, we examine:

- *collaboration based designs (CDBs)* as GenVoca components
- how to encode and statically compose components
- interesting case studies:
  - Graph Algorithm PLA
  - FSATS99 — Army Fire Support Simulator
  - Jakarta Tool Suite (JTS) — Extensible Java Compilers

Collaboration-Based Designs

Fundamental, but not well-known, technique for creating reusable OO designs

*Collaboration-Based Designs (CDBs)* idea:

- define a set of interrelated classes that collaborate to implement some “feature” or “aspect” of a program
- each class actually represents an individual role in that collaboration
- methods define generic interactions among class/roles

Graph Collaboration:

```
add_node()  belongs_to()
remove_node() linked_to()
graph_name() node_name()
roots()      connect_to()
...          ...
```
Collaboration Based Designs (Cont)

Note the following about CDBs:

- define generic relationships among classes/objects playing “roles”
  methods, instance variables, etc. that are needed to capture the desired relationships
  (in our case, node connectivity of a graph)

**whenever one is dealing with “Graph-Collaboration”
these methods will need to be written!**

CDBs generally aren’t stand-alone; that’s why roles exist

- *role is a parameter that must be instantiated*

Examples of ‘re’-using Graph-Collaboration:

- map plays role of graph, city corresponds to a node
- communication network is graph; site corresponds to a node

Problem with CDBs:

- not that well understood …
- known implementation techniques were not scalable (e.g., parameter instantiations of exponential length)
- no idea how they were related to component-based designs or refinements…

Let’s address these points first by examining how static refinements can be expressed in OO…
**OO Components**

How are static refinements expressed in OO?

- *refinement* of a class adds new data members, new methods, and/or overrides existing methods

  ![Diagram](image.png)

  expressed as a subclass

- a *GenVoca* or *large-scale refinement* adds new data members, methods, etc. simultaneously to several classes

  ![Diagram](image.png)

  note: there can be any number of “horizontal” or “collaboration” relationships among subclasses — here we show only inheritance relationships

**Insights**

Ever add a new feature to an existing application?

- changes aren’t localized!
- multiple classes of an application must be updated
- if feature is removed, updates must be simultaneously removed from all classes

A “feature” or “aspect” can be expressed as a CBD:

- consists of a number of collaborating classes/roles
- roles of a collaboration must be bound to classes of the actual application itself

  accomplished via parameter instantiation

  ![Diagram](image.png)

  *a layer defines a collaboration; layer instantiation defines role/class bindings!*

mixin-layers - 5
©dsbatory

mixin-layers - 6
©dsbatory
Composition Insights

When CBDs (GenVoca components) are composed, a forest of inheritance hierarchies is created that gets progressively broader and deeper!

L1
L2
L3
L4

note: $L_{i+1}$ inherits all the classes from $L_{i}$, recursively. Thus, the above instance of $L_4$ has a total of 5 classes, each the terminals of their refinement chains.

Composition Insights (Cont)

The classes that are instantiated by an application are the terminals of these refinement chains
- nonterminal classes define intermediate derivations of application classes

• in our example, “generated” application consists of 5 classes (shaded in black); the white classes are “intermediate” derivations of the black classes
1st Idea: Mixins

How can we encode refinements in OO languages?

*Mixin* is a class whose superclass is specified via a parameter

\[ M \text{<AnyClass S>} \text{ extends S { } } \]

Can express mixin \( M \) as Java “template” with parameter \( S \)

\[ \text{class M <AnyClass S> extends S { } } \]

- note: “mixin” means something different in C++, CLOS literature, so beware! We use Bracha’s definition…

2nd Idea: Nested (Inner) Classes

Nested (inner) classes behave (e.g., access control, scoping) like regular class members

```
class OuterParent { class Inner { } }
class OuterChild extends OuterParent { }
```

- no *OuterChild.Inner* explicitly defined, but it does exist … *Inner* is inherited from *OuterParent*

Nested classes emulate package encapsulation
- *representation allows “packages” to appear as nodes in inheritance hierarchies*
Combining Ideas = Mixin-Layers

Experience has shown that:

- different collaborations use the same names for roles
- classes with same role names refine each other when their collaborations are composed

Express as a mixin-layer $M$ with parameter $S$:

```java
class M <AnyClass S> extends S {
    class role1 extends S.role1 { ... }
    class role2 extends S.role2 { ... }
    ...
    class roleN extends S.roleN { ... }
}
```

Buy where do realms fit in?

Realms

Remember: realms have interfaces that components export and import

- so component $M[I]:E$ expressed as:

  ```java
  interface E { ... } // export interf.
  interface I { ... } // import interf.
  
  class M<I x> extends x implements E {
      class role1 extends x.role1 { ... }
      class role2 extends x.role2 { ... }
      ...
      class roleN extends x.roleN { ... }
  }
  // above is "standard" mixin-layer pattern
  ```
Connection to GenVoca

Straightforward connection to type equations:

```java
// type equation notation
Application = L4[ L3[ L2[ L1 ] ] ];

// extended-Java notation
class Application extends L4< L3< L2< L1 >>;
```

If we followed same derivation, except that objects are being refined, not classes, you’ll discover that OO frameworks are dynamic counterparts to Mixin-Layers

```java
// type equation notation
Application = L4[ L3[ L2[ L1 ] ] ];

// extended-Java notation
Application = new L4( new L3( new L2( new L1())));
```

• OO framework implements a realm of components, refinements...

Case Study: Graph Algorithm PLA

Look at a simple domain for which a PLA can be created
• to reinforce ideas just presented
• see “complex” example afterward (and in Appendix)

Domain: programs that implement different graph algorithms over directed and undirected graphs

• product line programs described by a “feature” menu:

<table>
<thead>
<tr>
<th>graph algorithm</th>
<th>search algorithm</th>
<th>graph type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex numbering</td>
<td>cycle checking</td>
<td>directed graph</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>undirected graph</td>
</tr>
<tr>
<td>depth-first</td>
<td>breadth-first</td>
<td>choose one</td>
</tr>
<tr>
<td>choose one or more</td>
<td>choose one</td>
<td>choose one</td>
</tr>
</tbody>
</table>

• example programs of product-line:
  - `ex1` — vertex numbering using a depth-first search on an undirected graph
  - `ex2` — vertex numbering and cycle checking using a breadth-first search on a directed graph
GenVoca Model

Define a realm $G$ of components that implement each individual “feature”:

\[
G = \{ \text{undirected,} \quad \text{directed,} \quad \text{dft}[x:G], \quad \text{bft}[x:G], \quad \text{cycle}[x:G], \quad \text{number}[x:G], \quad \ldots \}
\]

Composition restrictions (i.e. “choose one”, ordering) realized by design rules (discussed in next lecture)

Compositions (from previous page):

\[
ex1 = \text{number[ } \text{dft[ } \text{undirected } \text{] ]}
ex2 = \text{number[ } \text{cycle[ } \text{bft[ } \text{directed } \text{] ] ]}
\]

Implementation as Mixin-Layers

directed and undirected encapsulate two classes $\text{Graph}$ and $\text{Vertex}$

- methods support vertex addition and removal from graphs; no traversals

\[
\text{Graph} \quad \text{Vertex}
\]

dft and bft encapsulate a pair of refinements of $\text{Graph}$ and $\text{Vertex}$, and add new abstract class $\text{WorkSpace}$.

- traversal methods ($\text{GraphSearch}$, $\text{VertexSearch}$) added to $\text{Graph}$, $\text{Vertex}$ with a $\text{WorkSpace}$ object as parameter

- $\text{WorkSpace}$ object has 3 abstract methods: \text{init\_vertex}, \text{preVisitAction}, \text{postVisitAction}
Implementation (Cont)

cycle and number encapsulate refinements of Graph and Vertex (by adding algorithm-specific methods — ex: vertexNumber) and define a subclass of WorkSpace with appropriate init, pre-, post-action methods)

Example: `ex1 = number[ dft[ undirected ] ]`

Recap

Graph Domain illustrates the ability to:

- application of product-line defined by features it offers
- implement each feature as a mixin-layer

   each layer encapsulates multiple classes
   many domains have more complex features (e.g., features within features) — handled in a simple, parametric way rather than decomposing these features into a composition of layers…

- easy to implement in C++ and mixin-extended versions of Java

   see JTS example in Appendix of this lecture

- define product-line app as a composition of mixin-layers


Now let’s look at a more complex domain…
Case Study: FSATS99

*FSATS (Fire Support Automated Test System)*
- command-and-control simulator for Army fire support
- first-generation system (9 years to build)
- difficult to understand & maintain current code base
- difficult to debug
- new capabilities are projected, old capabilities revamped
- implementation team wants to expand capabilities, but not current version
- beginnings of a product-line (but no PLA yet)

Upcoming topics:
- explain domain of fire support
- quickly review current implementation of FSATS
- review component-based redesign (FSATS99) to see how mixin-layers were used to build an FSATS99 PLA
Vanilla Distributed Application

Set of collaborating objects that work collectively toward achieving a given mission

All sorts of different kinds of mission types:

- *when-ready-fire-for-effect* (WRFFE) - mortars
- *when-ready-fire-for-effect* - artillery
- *adjust-fire* (AF) - mortars
- *adjust-fire* - artillery
- about 20 mission types in all; more mission types projected in future

For each mission type, each OPFAC (e.g., FO, FIST, FSE, ...) takes different actions
- actions are coordinated with respect to particular mission
An OPFAC can be simultaneously processing any number of mission instances
- e.g., 2 WRFFE-mortars, 3 AF-artillery, etc.

Current Implementation

Is monolithic; each OPFAC is an Ada program that sends and receives tactical messages

When a message is received, it (and the current state of the OPFAC) is processed by a sequence of rules:

\[
\text{if (conditions}_1\text{) do-action}_1; \\
\text{if (conditions}_2\text{) do-action}_2; \\
\text{if (conditions}_3\text{) do-action}_3; \ldots
\]

Rules encode conditions (5-10 primitives) to fire actions for one or more missions

- 200-1000+ rules per OPFAC
- difficult to write, understand, debug rules
- hard to see what rules apply to given mission
- typical example of non-extensible system; started out small and understandable, but didn’t stay that way
FSATS99

Re-engineering FSATS to be a GenVoca PLA
• key idea: each Mission Type is a collaboration
  - Brigade
  - Fire Support Team (FIST)
  - Artillery
  - Forward Observer (FO)

• actions of each OPFAC is defined by a protocol (state machine) that it follows to do its part in processing an instance of a mission type

FSATS99 (Cont)

Each OPFAC may play a role in different mission types
• for each mission type, it has a protocol to follow to process instances of that type

thus, new code is added to process a mission
(whenever a new mission type is added to FSATS99)

vanilla
OPFACS

WRFFE-mtr
WRFFE-art
AF-mtr
AF-art

Most refined OPFAC classes “understand” capabilities and responsibilities of all mission types — only bottom classes are instantiated!
Advantages of Layered Design

Mission complexity is encapsulated in one spot — the layer definition

Mission types can be debugged separately, in isolation from each other
  • substantially simplifies mission development (over current version of FSATS)
  • using common OO design techniques to define collaborations (layers) — e.g., state machines — it is much easier to determine if all possibilities are accounted for

New mission types can be added/removed from FSATS99 through component composition/reconfiguration
  • extensible, creates PLA for FSATS simulators

Introspection

Could we have built FSATS99 in a different way, using standard OO techniques?

• ans: maybe, but generally no.

Insights:
  • OO design methodologies look for objects, classes, and their relationships; GenVoca seeks the fundamental refinements in a domain — find them first, choose their implementation later

  • OO design methodologies direct you toward the design of the most refined classes of an application, and do not expose the intermediate derivation classes. That is, an application is one big collaboration, instead of a composition of smaller reusable collaborations

  • want a higher-level ability to specify applications as type equations, rather than writing code. We want to program at the architectural level, not code level.
GenVoca Again

FSATS99 PLA model has single realm; all components
refine basic OPFAC abstraction

\[ F = \{ \text{vanilla,} \quad \text{// basic opfacs} \]
\[ \text{wrffe-common}[F], \quad \text{// shared by wrffe} \]
\[ \text{wrffe-mtr}[F], \]
\[ \text{wrffe-art}[F], \]
\[ \text{af-common}[F], \quad \text{// shared by af} \]
\[ \text{af-art}[F], \]
\[ \text{af-mrt}[F], \]
\[ ... \]

Compositions yield FSATS simulator. Ex:

\[ \text{fsats101} = \]
\[ \text{wrffe-mtr[ af-art[} \]
\[ \text{wrffe-common[ af-common[} \]
\[ \text{vanilla ] }] ] }; \]

Big Picture

Graph Algorithm PLA, FSATS99, JTS aren’t isolated examples. Similar GenVoca architectures exist for:

- avionics (ADAGE)
- database systems (Genesis)
- network protocols (Ensemble)
- ...

Graph PLA, FSATS99, JTS components are mixin-layer (template-based representations) of refinements.

Remember, there are lots of other representations!
Conclusions — Lessons Learned

- collaboration-based designs correspond to GenVoca layers or aspects; compositions of CBDs correspond to GenVoca type equations

  connection with important and under-appreciated OO design technique

- a mixin-layer is a template-based construct that implements a GenVoca refinement

  novel mixture of parameterized inheritance and nested classes;

  mixin-layers provide scalable implementations of collaboration-based designs

- case studies: FSATS99, JTS (appendix)

  nontrivial examples of scalable PLAs using mixin-layers

Further Readings


**Appendix: JTS Case Study**

**JTS (Jakarta Tool Suite)**

- compiler tool suite to create extensible Java languages
- motivation: need tool suite to help write domain-specific languages and domain-specific extensions to host programming languages

```java
// ex #1 - metaprogramming addition to Java
AST_Exp x = exp{ q > z }exp;
AST_Stm s = stm{ if (x.alpha()) foo();
    else bar(); }stm;
```

```java
// ex #2 - type equations to extend Java
class Application extends L4< L3< L2< L1 >>>;
```

- want a product-line of Java dialects where each optional feature encapsulated as a component; don’t want to build monolithic precompilers, as there would be an exponential number of them
- particular dialect specified by composition of components of desired features

**How JTS Works**

Programs are represented as parse trees (syntax trees)

- parse trees are infinitely extensible

Microsoft IP’s “Intentions”

- add AST nodes with domain-specific semantics

ex: AST constructors, type equations

- at reduction time, intention nodes are replaced with their Java (or host language) implementations

ex: replace code constructors, type equations with their pure Java counterparts
How are lexers, parsers, and transform programs created?
• ans: Bali tool

Bali is a GenVoca generator of (Java) preprocessors

• assembles variants of Jak (extensible Java) from components

• components encapsulate primitive Java extensions

ex: AST constructors
    hygienic macros
    P3 data structure generators
    layer definition and composition etc.

A Bali component is an ordered pair (syntax, semantics)

• syntax — grammar extensions to Java

• semantics — meaning given to grammar extensions

Bali Syntax

Are extended, annotated BNF grammars

• extended with repetitions (see POPART)

\[
\text{StatementList} : ( \text{Statement} )^+;
\]

\[
\text{ArgumentList} : \text{Argument} ( ',', ' Argument ' )^*;
\]

• annotated by class to instantiate when production is recognized (e.g., see POPART)

\[
\text{SelectionStmt}
\quad : \text{IF} \ '( ' Expr ' )' \ \text{Statement} :: \text{IfStm}
\quad | \text{SWITCH} \ '( ' Expr ' )' \ \text{Block} :: \text{SwitchStmt};
\]

from grammar, can infer constructors:

\[
\text{IfStm( token, token, AST_Exp, token, AST_Stm )}
\]

\[
\text{SwitchStmt( token, token, AST_Exp, token, AST_Blk )}
\]
Inheritance Hierarchies

Can be deduced from grammar specifications:

\[
\begin{align*}
\text{Rule1} & : \text{pattern1} :: \text{C1} \\
& | \text{Rule2} \\
\end{align*}
\]

\[
\begin{align*}
\text{Rule2} & : \text{pattern2} :: \text{C2} \\
& | \text{pattern3} :: \text{C3} \\
\end{align*}
\]

Grammar specifications used for:

- defining host grammar (e.g. Java)
- defining additional rules, lexical tokens for grammar extensions

Bali Grammar Specifications

What can be generated automatically:

- lexical analyzer
- parser — now using JavaCC
- inheritance hierarchy and AST classes
- class constructors, unparsing, tree editing methods
What Can’t be Generated?

Type checking, reduction, and optimization methods

- AST node specific
- hand code as subclasses to Bali-generated classes

Bali generates a mixin layer that encapsulates all generated AST classes

- separate mixin-layer encapsulates the hand-coded subclasses of each class defined in a Bali-grammar file

Relationship to GenVoca/Mixin-Layers

Notes:
- terminals of refinement chains are the classes that are instantiated
- type equation to scale:

```java
```

- inheritance hierarchy not drawn to scale: 500+ classes, some mixin layers have over 100 classes each
GenVoca Again

JTS PLA model for Java dialects has 2 realms:

\[
K = \{ \text{kernel} \} \\
J = \{ \text{javal}_0[K], \quad \text{// Java 1.0} \\
\text{javal}_1[J], \quad \text{// Java 1.1 ext to 1.0} \\
\text{AST}[J], \quad \text{// metaprogramming} \\
\text{gscope}[J], \quad \text{// hygienic macros} \\
\text{layerdef}[J], \quad \text{// layer definitions} \\
\text{Teqn}[J], \quad \text{// type equations} \\
\ldots
\}
\]

Compositions yield a Java dialect. Ex:

```
JavaPlusPlus =
    Teqn[ layerdef[
        AST[ javal_1[javal_0[
```
Design Rule Checking

Not all syntactically correct combinations of GenVoca components are semantically correct. Some components work only in the presence (or absence) of other components

- fundamental problem: impossible for generator users to debug generated code; need automated help to debug component compositions
- design rules are domain-specific constraints that specify illegal configurations of components. Design rule checking (DRC) is the process of (automatically) applying design rules

In this lecture, we present:

- a model of DRC based on attribute grammars
- has been used in every GenVoca PLA
- relate DRC to research on software architectures

Motivating Example: P3

Generator for container data structures

- relies on two realms containing 50 components:

  ```
  ds = { bintree[ ds ],       // binary tree
dlist[ ds ],            // unordered list
olist[ ds ],            // ordered list
avail[ ds ],            // free list manager
array[ mem ],           // sequential storage
malloc[ mem ],          // random storage
inbetween[ ds ],        // common delete code
... }
  mem = { transient,      // in memory storage
           persistent,    // memory mapped
            }
  ```

Data structures are modeled by type equations

- reference 5 to 15 components
- too elaborate to validate by inspection
- some components have obscure rules for their use
Example P3 Design Rule

`inbetween` component encapsulates:
- algorithms shared by many data structure components (e.g., `bintree` and `dlist`)
- deals with positioning of cursor after element is deleted
- details complex, hidden from user

Correct usage of `inbetween` requires:
- one copy in TE that has 1+ data structure components &
- precede all such components in equation
  ```
  right = ...inbetween[ ... dlist[ bintree[ ... ]]];
  wrong = ... dlist[ inbetween[ bintree[ ... ]]];
  ```

Such rules should not be borne by programmers
- too easy to forget and be misapplied

> want rules tested automatically

Results from Software Architectures

Perry’s *Inscape* (1989) is environment for managing evolution of software systems:
- novel aspects: obligations and consistency checking
  - *light semantics*

Components have pre-, post-conditions, and obligations

*bank loan example*

- *obligations* are conditions that must be satisfied by system that uses the component
- require “action-at-a-distance” — nonlocally satisfied
- propagated to enclosing modules where they are eventually satisfied by some postcondition

*Full-fledged verification not attempted*

- primitive predicates declared (but informally defined)
- pre-, post-, obligations expressed in terms of primitives
- practical & powerful form of “shallow” consistency checking using pattern matching and simple deductions
Design Rule Checking

Adapt and generalize Inscape consistency checking to DRC by exploiting the semantics of GenVoca layers

(1) DRC models states of system (TE) design
- not states of system execution
- model states / properties of system design by assigning values to attributes
- exploit refinement interpretation of layers

DRC Basics

(2) Preconditions and obligations of component K are satisfied “at-a-distance” by components that lie either:
- (far) beneath K or
- (far) above K

constraints typically not satisfied by adjacent components (c.f. Goguen, Tracz, Sitaraman references)

Properties exported to “higher” layers generally not the same as properties exported to “lower” layers

Leads to 2 kinds of design rules:
- #1: preconditions for component usage
  ```
  \begin{array}{c}
  K \\
  \text{pre: } A = v \\
  \text{post: } A = v
  \end{array}
  ```
DRC Basics (Continued)

• #2: preconditions for parameter instantiation

new names: preconditions called prerestrictions
postconditions called postrestrictions

note: prerestrictions correspond to Inscape obligations

Components have:

Design rule checking involves:
• top-down propagation of postconditions and testing component preconditions
• bottom-up propagation of postrestrictions and testing of parameter prerestrictions

In following, we assume no restriction on complexity of predicates, but will later show that very simple predicates suffice for P3 (and other domains as well).
**Top-Down DRC**

Components have preconditions and postconditions

```
S
  A
  top ⇒ precondition-A
  postcondition-A ⊕ top = top'

B
  top' ⇒ precondition-B
  postcondition-B ⊕ top' = top''

C
  top'' ⇒ precondition-C
  postcondition-C ⊕ top''
```

- postconditions propagated by ⊕ operator
- conditions tested by ⇒ operator

**A Twist...**

Consider component with multiple parameters: A[x,y]

- gives rise to trees (dags)
- twist: each parameter has its own postcondition

```
precondition-A

A
  x
  y
  postcondition-Ax
  postcondition-Ay
```

i.e., conditions for parameter x may be different for those of y in A[x,y]

- example: the realm of a parameter could be expressed as a postcondition; realms for x and y could be different
Top-Down Algorithm

Top-Down Algorithm is a simple recursive algorithm for top-down propagation of conditions and testing component preconditions.

```
<table>
<thead>
<tr>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>post-A + top = top-A</td>
</tr>
<tr>
<td>post-Bx + top-A = top-Bx</td>
</tr>
<tr>
<td>post-C + top-Bx</td>
</tr>
<tr>
<td>post-D + top-By</td>
</tr>
</tbody>
</table>
```

Bottom-Up DRC

Conditions must also be propagated upwards...

- parameters of components have **prerestrictions** for instantiations to be correct

```
<table>
<thead>
<tr>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
<tr>
<td>w</td>
</tr>
</tbody>
</table>
```

*systems instantiate parameters, not components*

- exported states (called postrestrictions) propagated upwards so that prerestrictions can be tested

**prerestrictions for c are generally not satisfied by the component x that instantiates its parameter, but rather by components deep within the system rooted by x**
**Bottom-Up DRC Continued**

Every component has *postrestrictions*, i.e., exported states, and *prerestrictions* for each parameter

- **top**: set of required properties
  
  \[ \uparrow \]
  
  \[ \text{postrestriction-}A \oplus \text{bot}' \]

- **prerestriction-A**
  
  \[ \uparrow \]
  
  \[ \text{postrestriction-}B \oplus \text{bot} = \text{bot}' \]

- **prerestriction-B**
  
  \[ \uparrow \]
  
  \[ \text{postrestriction-}C = \text{bot} \]

- **A**
- **B**
- **C**

- use same operators \( \oplus \) and \( \Rightarrow \) for bottom-up DRC
- simple recursive algorithm for bottom-up DRC

---

**Attribute Grammars**

McAllester observed *attribute grammars* unify realms, attributes, top-down & bottom-up DRC algorithms

- realms of components modeled by grammars
- attributes model program development states
- postconditions are *inherited attributes* (values determined by ancestors)
- postrestrictions are *synthesized attributes* (values determined by descendants)

**Bonus:** common tools (lex, yacc) well-suited for implementing design rule checkers

Need to supply definitions & representations for:

- attributes
- predicates
- operators \( \oplus \) and \( \Rightarrow \)

For the P3 and Genesis generators…
P3 Attributes

Each attribute models a property that exposes a composition constraint
Attributes have restricted values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>nothing is known about property</td>
</tr>
<tr>
<td>assert</td>
<td>property is asserted</td>
</tr>
<tr>
<td>negate</td>
<td>property is negated</td>
</tr>
<tr>
<td>inherit</td>
<td>attribute value inherited, but is otherwise unconstrained</td>
</tr>
</tbody>
</table>

- example:
  attribute: component_belongs_to_realm_A
  attribute value: assert (or negate)

- see Batory and Geraci *IEEE TSE 1997* paper
  (and report UTCS TR-94-03 for other values…)

P3 Predicates

Preconditions & prerestrictions request specific attribute values (any, assert, negate), but not how they were determined (i.e., inherit)

- only 4 different *primitive* predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-any</td>
<td>true (no constraints)</td>
</tr>
<tr>
<td>P-assert</td>
<td>attribute has assert value</td>
</tr>
<tr>
<td>P-negate</td>
<td>attribute has negate value</td>
</tr>
<tr>
<td>P-false</td>
<td>false (unsatisfiable)</td>
</tr>
</tbody>
</table>

- complex predicates are typically conjunctions of primitives, one primitive predicate for each attribute
- encode as vector of predicates indexed by attribute

\[ P \equiv P_1 \land P_2 \land \ldots \equiv [ P_1, P_2, \ldots ] \]
Postcondition Propagation Operator \( \oplus \)

Component postconditions assert or negate values, or may propagate values

- table below defines the condition propagation operator + for a single attribute:

<table>
<thead>
<tr>
<th>component postcondition + existing condition</th>
<th>component postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>existing condition</td>
<td>assert</td>
</tr>
<tr>
<td>any</td>
<td>negate</td>
</tr>
<tr>
<td>assert</td>
<td>negate</td>
</tr>
<tr>
<td>negate</td>
<td>assert</td>
</tr>
</tbody>
</table>

- given \( P = [ P_1, P_2, ... ] \) and \( E = [ E_1, E_2, ... ] \)

\[
P \oplus E = [ P_1 + E_1, P_2 + E_2, ... ]
\]

Implication Operator \( \Rightarrow \)

The implication operator \( \rightarrow \) for a single attribute is:

<table>
<thead>
<tr>
<th>Existing Condition</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>P-assert</td>
</tr>
<tr>
<td></td>
<td>P-negate</td>
</tr>
<tr>
<td></td>
<td>P-false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Existing Condition</th>
<th>P-any</th>
<th>P-assert</th>
<th>P-negate</th>
<th>P-false</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>negate</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>any</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

- given \( E = [ E_1, E_2, ... ] \) and \( P = [ P_1, P_2, ... ] \)

\[ E \Rightarrow P = ( E_1 \rightarrow P_1 ) \wedge ( E_2 \rightarrow P_2 ) \wedge ... \]
Implementation Notes

Straightforward implementation: 1500 lines in lex & yacc

DRC algorithm is efficient: \( O(mn) \)

\( m = \# \) of attributes, \( n = \# \) of components

Example domain models:

<table>
<thead>
<tr>
<th>Generator (Domain)</th>
<th># of Realms</th>
<th># of Components</th>
<th># of Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genesis (databases)</td>
<td>9</td>
<td>52</td>
<td>14</td>
</tr>
<tr>
<td>JTS (Java precompilers)</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>P3 (data structures)</td>
<td>2</td>
<td>50</td>
<td>7</td>
</tr>
</tbody>
</table>

Some P3 attributes:

<table>
<thead>
<tr>
<th>attribute</th>
<th>property description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical_deletion</td>
<td>“a logical deletion layer”</td>
</tr>
<tr>
<td>retrieval</td>
<td>“a retrieval layer”</td>
</tr>
</tbody>
</table>

Straightforward Specifications

Example component & design rule declaration:

```plaintext
array : ds [ mem ] {
    # logical del. layer required above array
    precondition assert logical_deletion
    # assert that array is a retrieval
    # layer to all descendants and ancestors
    postcondition assert retrieval
    postrestriction assert retrieval
}
```
Explanation Based Error Reporting

In addition to detecting errors, we can extend DRC algorithms to suggest how to repair a type equation
- *precondition ceilings* from Inscape:

- error located in between components X and Y
- similar technique for obligations/prerestrictions

Example

Example: want container implementation that stores elements onto a binary tree, whose nodes are stored sequentially in transient memory

First attempt at composition:

```plaintext
first = top2ds[bintree[array[transient]]];
```

DRC response:

**Precondition errors:**
- an inbetween layer is expected between top2ds and bintree
- a logical deletion layer is expected between top2ds and array

**Prerestricion error:**
- parameter 1 of top2ds expects a subsystem with a qualification layer
**Explanation Based Error Reporting**

Clumsy fix…

```
second = top2ds[ inbetween[ bintree[ qualify[
            delflag[ array[ transient ]]]]]];
```

DRC response:

Precondition error:
- a retrieval layer (bintree) not expected above qualify

Correct type equation — swap qualify with bintree:

```
third = top2ds[ inbetween[ qualify[ bintree[
            delflag[ array[ transient ]]]]]];
```

---

**Insights**

Why isn’t DRC a challenging problem in program verification?
- solution unlikely to be automatable

Inscape work and our work have observed:
- problem is straightforward
- solution automatable and efficient

… why? … 2 reasons

Reason #1:
- shallow consistency checking goes a very long way
- in general, most logical errors are shallow errors

conjecture: all errors at component composition level should be shallow

- remaining errors must be dealt with by component implementors
Insights (Cont)

Reason #2: important distinction:
• Inscape components are functions
• GenVoca components are subsystems

Large applications consist of tens of thousands of lines of code
• hundreds or thousands of functions
  ⇒ hundreds or thousands of primitive predicates
• TEs rarely have more than 50 components
  ⇒ modest # of primitive predicates in a domain ~10-40
    seems counterintuitive

Why?
• modeling states of development (not execution) reduces number of properties to examine
• and GenVoca is a methodology for designing reusable components ...

The Key

What makes OO designs so powerful and attractive?
• Ans: ability to manage and control software complexity

Standardization is powerful way of managing and controlling software complexity in product-line architecture

Standardization makes some problems tractable that would otherwise be very difficult

• ex: composing off-the-shelf components
• composition of components is simple in GenVoca
• standardization seems to limit the ways in which components can constrain each other’s behavior
  ⇒ make DRC tractable

Historical perspective…
Additional Insights

We understand software in terms of implementation-independent refinements

- enhances power of design rule checking

- DRC tells you whether two refinements (concepts) can be composed *regardless how they are implemented*

  ex: `bintree[ encrypt[...] ]` may be correct
  ex: `encrypt[ bintree[...] ]` is always incorrect

- design rules allow you to state whether certain combinations of *concepts or features* (i.e., refinements) are possible

Conclusions — Lessons Learned

- **GenVoca domain models (realms + design rules) are attribute grammars**

  can use existing tools (lex, yacc) to express models

- **simple, efficient algorithms for DRC**

  constraints imposed on higher layers (preconditions)
  constraints imposed on lower layers (prerequisites)
  don’t need formal methods, theorem provers

- **components that are designed to be interoperable, plug-compatible, and interchangeable often makes complex problems much easier to solve**

  standardization of programming abstractions is a powerful way of controlling the complexity of a product-line (i.e., family of systems)
Further Reading


Further Reading (Continued)


Reading and Reference List

1 Background

1.1 Historical Precedence

McIlroy was among the first to identify the problem of library scalability; the notion of virtual machines is due to Dijkstra, and families of systems is due to Parnas.


1.2 Parameterized Programming

The concepts of horizontal parameterization (i.e., parameterizing interfaces) and vertical parameterization (i.e., layering) have been expressed elegantly by Goguen and Tracz. The certification of parameterized components has been examined in the RESOLVE project.


1.3 Large System Development

References that survey the problems of large system development (and indirectly, problems that arise in domain modeling) are:


1.4 Object-Oriented Frameworks

GenVoca components encapsulate suites of interrelated classes. So too do object-oriented frameworks; they are suites of interrelated abstract classes that have multiple concrete class implementations, which describe different implementations of what we have called “subsystems”. A framework is an OO way of representing a realm of components; the abstract classes define both the interface of a realm and code that is common across all components, whereas concrete subclasses provide component-specific implementations. What frameworks lack are parameterizations and method wrappers that are needed to express GenVoca compositions (See Subjectivity and Method Wrappers).


1.5 Role-Based Designs and Mixins

Role-based designs is a design technique that encapsulates features of applications through a set of classes that perform specific roles. Role-based designs are an object-oriented design technique that can be used to design GenVoca layers. Implementing role-based designs is through the use of mixins, classes whose superclass are specified via a parameter. The following papers survey recent work on mixins and on role-based designs.


1.6 Domain Modeling

A key problem in modeling domains is choosing the right abstractions. The following references offer a variety of practical perspectives on this topic. (See also papers on Frameworks, and Role-based Designs and Mixins).


1.7 Software Reuse

Good overviews of the state-of-art results and problems in software reuse are:


1.8 Transformation Systems

GenVoca domain models can be implemented by program transformation systems. Simonyi’s paper describes an innovative approach to the integration of transformation systems and compilers to create extensible programming languages. Weigert’s paper describes an impressive transformation system that is being used at Motorola
to produce customized software for different radio-products. Griswold’s paper deals with semantics preserving program transformations. The papers by Roberts, et al and Tokuda et al describe innovative approaches to creating tools for editing OO programs by transformations. Pu’s paper is an example of a dynamic generative generator, and the other papers deal with transformations in the context of “parameterized layers” of rewrite rules:


1.9 Software Architectures
Software architectures deal with issues of “programming-in-the-large” and building software systems from components. A variety of popular perspectives are:
E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

1.10 Product-Line Architectures
A natural out-growth of research on architectures is product-line architectures. The idea of building families of applications isn’t new (see Parnas) nor is building families of applications from components (that’s GenVoca, Draco). However, some fresh ideas have begun to surface. You may want to consult the Proceedings of the 1st Software Product-Lines Conference, Denver, Colorado in August 2000.
1.11 Subjectivity

When modeling families of related applications, objects do not have single interfaces. Rather, they are described by families of related interfaces. The interface that is appropriate for an application is application specific (i.e., subjective). The following references give an overview of current thinking on the topic. Also see Role-Based Designs and Mixins.


1.12 Component Design

The following papers give an overview of techniques on how components can be implemented. Please refer to the Subjectivity lecture for an overview. (See also Method Wrappers and Role-based Designs and Mixins).


M. Van Hilst and D. Notkin, in Subjectivity above.

Y. Smaragdakis and D. Batory, in Subjectivity above.

1.13 Method Wrappers

The encapsulation of method wrappings within components is an important part of the GenVoca model. The following references provide a state-of-the-art look at current ideas in method wrappings:


1.14 Validating Component Compositions

Shallow consistency checking is an important technique used in software architectures and software system generators in order to validate compositions of components. It offers a practical approach to consistency checking without requiring full-fledged verification.

1.5 Scalability

Papers that put forth similar arguments for library scalability are:


1.6 Generators

GenVoca is not the only way in which software generators are implemented. Other approaches include:


H. Gomaa, L. Kerschberg, et. al., paper in Domain Modeling (above).


2 GenVoca

The first paper covers many of the basic concepts of GenVoca. The second paper focuses on composition validation and subjectivity. The third paper presents a broad overview of results and experiences using GenVoca generators. The “Design Wizards” paper shows how type equations can be optimized. The Czarnecki and Eisenecker paper gives a very good overview of the goals of layering, software synthesis, and product-lines.


Also see http://www.cs.utexas.edu/users/schwartz/index.html

2.1 ADAGE (Avionics)


2.2 Avoca/x-kernel (Communication Networks)


Also see: http://www.cs.arizona.edu/xkernel/www/index.html

2.3 Ficus (File Systems)


Also see http://www.isi.edu/~johnh/WORK/index.html

2.4 Genesis (Database Management Systems)

Reading and Reference List


Also see http://www.cs.utexas.edu/users/schwartz/index.html

2.5 P2 and P3 (Data Structures)


Also see http://www.cs.utexas.edu/users/schwartz/index.html

2.6 Jakarta Tool Set


Also see http://www.cs.utexas.edu/users/schwartz/index.html