Feature Modularity for Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Copyright is held by the author/owner(s).
Presented at: OOPSLA '06
Generative Programming and Component Engineering (GPCE)
Portland, Oregon, USA
October 22-26, 2006
2006 ACM 06/0010
Feature Oriented Programming for Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Introduction

A **product-line** is a family of similar systems
- Chrysler mini-vans,
  Motorola hand-held radios,
  software

**Motivation: economics**
- amortize cost of building variants of program
- design for family of systems

**Key idea of product-lines**
- members of product-line are differentiated by features
- **feature** is product characteristic that customers feel is important in describing and distinguishing members within a family
- **feature** is increment in product functionality

Very Rich Technical Area...

- **Feature Oriented Programming (FOP)** is the study of feature modularity in product-lines
  - features are first-class entities in design
  - often implemented by collaborations, crosscuts

- History of applications
  - 1986 database systems
  - 1989 network protocols
  - 1993 data structures
  - 1994 avionics
  - 1997 extensible Java compilers
  - 1998 radio ergonomics
  - 2000 program verification tools
  - 2002 ExCIS fire support simulator
  - 2003 AHEAD tool suite
  - 2004 robotics controllers

- **Integrates many subjects:**
  - compilers
  - grammars
  - artificial intelligence
  - databases
  - algebra
  - category theory
  - programming languages
  - compositional programming & reasoning
  - OO software design
  - aspect-oriented programming
  - others...

©dsbatory intro-1

©dsbatory intro-2

©dsbatory intro-3

©dsbatory intro-4
Tutorial Overview

- Part I
  - The FOP Paradigm
  - The Theory
  - AHEAD Tool Suite

- Part II
  - Verification of Feature Compositions
  - Feature-Oriented Model Driven Development

The FOP Paradigm

a general approach to program
development and product-line synthesis

Motivation

- Software products are:
  - increasing in complexity
  - increasing in costs to develop and maintain
  - decreasing in ability to understand

- Basic goal of SE is to manage and control complexity
  - structured programming to
  - object oriented programming to
  - component-based programming to...

- today’s design techniques are too low-level,
  exposing too much detail to make application’s design,
  construction and modification simple

- Something is missing...
  - future design techniques generalize today’s techniques
  - tutorial to expose a bigger universe

Keys to the Future

- New paradigms will likely embrace:
  - Generative Programming (GP)
    - want software development to be automated
  - Domain-Specific Languages (DSLs)
    - not Java & C#, but high-level notations
  - Automatic Programming (AP)
    - declarative specs → efficient programs

- Need simultaneous advance in all three fronts to
  make a significant change
Not Wishful Thinking...

- Example of this futuristic paradigm realized over 25 years ago
  - around time that AI researchers gave up on automatic programming

Relational Query Optimization

Keys to Success

- Automated development of query evaluation programs
  - hard-to-write, hard-to-optimize, hard-to-maintain
  - revolutionized and simplified database usage
- Used algebra to specify and optimize query evaluation programs
- Identified fundamental operations of this domain
  - relational algebra
- Represented program designs as expressions
  - compositions of relational operations
- Define algebraic identities among operations to optimize expressions
- Compositionality is hallmark of great engineering models

Looking Back and Ahead

- Query optimization (and concurrency control) helped bring DBMSs out of the stone age
- A Holy Grail Software Engineering:

  Repeat this success in other domains

- Not obvious how to do so...
- Subject of this tutorial...

  - series of simple ideas that generalize notions of modularity and lay groundwork for practical compositional programming and an algebra-based science for software design
Today’s View of Software

- Today’s models of software are too low level
  - expose classes, methods, objects as focal point of discourse in software design and implementation
  - difficult (impossible) to
    - reason about construction of applications from components
    - produce software automatically from high-level specifications (distance is too great)
- We need a more abstract way to specify systems

A Thought Experiment...

- How do you describe program that you’ve built...
  - don’t say which DLLs are used...
- Instead, say what features your program offers its clients
  - Program1 = feature_X + feature_Y + feature_Z
  - Program2 = feature_X + feature_Q + feature_R
    - why? because features align better with requirements
- We should specify systems as compositions of features
  - few do this for software now
  - done in lots of other areas

Dell Web Site

Declarative DSL to select features of desired system
Methodology for Construction

- What methodology builds systems by progressively adding details?

- **Stepwise Refinement**
  - Dijkstra, Wirth early 1970s
  - abandoned in early 1980s as it didn’t scale...
  - had to compose hundreds or thousands of transforms (rewrites) to produce admittedly small programs
  - recent work shows how SWR scales
    - scale individual transform to a feature
    - composing a few refinements yields an entire system

**Terminology Disclaimer**

- We use OO meaning of term “refinement”
  - elaboration of an object (objects) that introduces a new service, feature, or relationship

- In algebraic communities
  - “refinement” means add detail, but **no** new capability
    - e.g., implement an interface
  - our use of ‘refinement’ is ‘extension’ in algebraic communities
  - “stepwise development”

- Use terms “extension” and “refinement” interchangeably
  - although, beware of different meanings!

**What is a Feature?**

- **Feature**
  - an elaboration or augmentation of an object(s) that introduces a new service, capability, or relationship
  - increment in functionality

- **Characteristics**
  - abstract, mathematical concept
  - reusable
  - interchangeable
  - (largely) defined independently of each other

- Illustrate in next few slides
Tutorial on Features (Refinements)

Features are Interchangable

Features are Interchangable
Features are Interchangable

Features are Reusable

Features are Functions!

Composing Features

- Feature composition = function composition

$$= \text{lincolnBeard}(\text{uncleSam}(\text{mustache}(\text{beanie}(\text{PersonPhoto}\ x))))$$
Large Scale Features

- **Called Collaborations (1992)**
  - simultaneously refine multiple objects/entities
  - refinement of single entity is called role
  - crosscuts

- **Example: Positions in US Government**
  - each defines a role

  Prez  Vice Prez  ....

Composing Collaborations

- At election-time, collaboration remains constant, but objects that are refined are different

  Prez  Vice Prez

  Example of dynamic composition of collaborations

Other Collaborations

- **Parent-Child collaboration**

  Parent  Child

- **Professor-Student collaboration**

  Prof  Student

Example

- Kelly  Steve  Don  Alex  Mark

  Prof  Student

  Parent  Child

  Parent  Child
Same Holds for Software!

Highly complex entities and relationships in software can be synthesized by composing generic & reusable features

Part I: The Theory

GenVoca and AHEAD

Next Generation in Last Lecture

Feature Oriented Programming

- Feature Oriented Programming (FOP) is the study of feature modularity and programming models for product-lines
  - a powerful form of FOP based on stepwise development
  - advocates complex programs constructed from simple programs by incrementally adding features
- How are features and their compositions modeled?

A Clue...

- Consider any Java class C
  - member could be a data field or method
  - class C below has 4 members m1–m4

```java
class C {
    member m1;
    member m2;
    member m3;
    member m4;
}
```
Have You Ever Noticed…

- Contents of C can be distributed across an inheritance hierarchy

```
class C1 {
    member m1;
}

class C23 extends C1 {
    member m2;
    member m3;
}

class C4 extends C23 {
    member m4;
}

class C extends C4 {}
```

Another Example...

- C23 decomposed further as:

```
class C2 extends C1 {
    member m2;
}

class C3 extends C2 {
    member m3;
}

class C23 extends C3 {}
```

Observe…

- Significance: a class definition need not be monolithic, but can be built by incrementally composing reusable pieces via inheritance

- Nothing special about the placement of members m1…m4 in this hierarchy except...

  - **no-forward references**: member can be introduced as long as all members it references are defined

- requirement for compilation, stepwise development

Look Familiar?? Remember Algebra?

- Consider sets and union operation (∪)
  - commutative almost like inheritance

```
C1 = { m1 }
C2 = { m2 }
C3 = { m3 }
C4 = { m4 }
C = C1 ∪ C2 ∪ C3 ∪ C4
   = { m1, m2, m3, m4 }
```

- Vector addition (+)
  - is commutative almost like inheritance

```
C1 = (m1,0,0,0)
C2 = (0,m2,0,0)
C3 = (0,0,m3,0)
C4 = (0,0,0,m4)
C = C1 + C2 + C3 + C4
   = ( m1, m2, m3, m4 )
```
A Closer Analogy

- Vector join (→)
- Vector join lays vectors end-to-end to define a path
- Not commutative! – Order of composition matters!

\[
\begin{align*}
C_1 &= (m_1,0,0,0) \\
C_2 &= (0,m_2,0,0) \\
C_3 &= (0,0,m_3,0) \\
C_4 &= (0,0,0,m_4)
\end{align*}
\]

\[
A \rightarrow B \neq B \rightarrow A
\]

Operation We Want...

- Is not inheritance...
  - want to add new methods, new fields, and refine existing methods like inheritance
  - also want constructors to be inherited and static methods refined as well; Java inheritance doesn’t provide this

```
class C {
    constructor#1
    constructor#2
}
```

The operation • we want is called **class refinement**

Syntax of Class Refinement

- Suppose program P has single class B
- Composition of R with P defines a new program N:

```
class B {
    int x;
}
```

```
class B {
    int x;
    int y;
    void z(){...}
}
```

A refinement R adds y, z()

```
refines class B {
    int y;
    void z(){...}
}
```

Algebraic Formulation

- Base programs are **constants**

```
// constant P
class B { int x; }
```

```
// function R
refines class B {
    int y;
    void z(){...}
}
```

- Composition is an **expression**

```
N = R( P )
```

```
class C {  
    constructor#1  
    constructor#2  
}
```

```
class C {  
    constructor#1  
    constructor#2  
}
```

The operation • we want is called **class refinement**

```
Treat programs as values
```
Another Example

```
class C { member m1; }    // constant C1
refines class C { member m2; }    // function C2
refines class C { member m3; }    // function C3
refines class C { member m4; }    // function C4
```

- Composition is an **expression** or named expression

```
C = C4( C3( C2( C1 ) ) )  // constant C1
    = C4 • C3 • C2 • C1
```

Note: both notations are equivalent

Method Refinement

```
result = method_refinement

void foo() {
   /* before stuff */
   super.foo();
   /* after stuff */
}
```

- base_method

```
void foo() {
   /* before stuff */
   /* do something */
   /* after stuff */
}
```

(or an equivalent encoding)

A Quick Note on Aspects

- If aspects are **functions** that map programs, you could use aspects to implement refinements described here and more
  - cflows, etc.

- But beware! Without basic changes, advice in AspectJ does **not** have functional semantics
  - can create unpredictable results
  - recent surveys of usage of AspectJ suggests about 90 percent of advice & introductions correspond to implementations of collaborations that we discuss

Connecting the Dots...

- **Scalability**
  - effects of refinement not limited to a single class
  - **collaborations** encapsulate refinements of multiple classes as well as adding new classes
    - adding new classes that can be refined is critical
Connecting the Dots...

- A **collaboration** has meaning when it implements a feature
  - ever add a new feature to an existing OO program?
  - several classes must be refined as well as adding new classes
  - crosscuts

Program Synthesis Paradigm

Program $P =$ featureZ ● featureY ● featureX

By composing features, packages of fully-formed classes are synthesized

Contributors to this View…

- Many researchers have variants of this idea:
  - refinements - Dijkstra, Wirth 68
  - layers - Dijkstra 68, Batory 84
  - product-line architectures - Kang 90, Gomaa 92…
  - collaborations - Reenskaug 92, Lieberherr 95, Mezini 03
  - program verification - Boerger 96
  - aspects - Kiczales 97, et al.
  - concerns - Ossher-Harrison-Tarr 99

Connecting the Dots...

- You can always decompose software in this manner
  - trick is that your refinements be reusable
  - that's the connection with features, product-lines
  - features are reusable – so too must be their implementations

- software that is not designed to be reusable, composable, etc. with other software won't be – this is **co-design** or designing to a **standard**
- **Architectural Mismatch** (ICSE 1995)

- **Product-line design** – feature implementations are designed with compositionality, reusability in mind
GenVoca (1988,1992)

- Equates constants, functions with features
- A domain model or product-line model or GenVoca model M

- Constants:
  - f – base program with feature f
  - h – base program with feature h

- Functions
  - i • x – adds feature i to program x
  - j • x – adds feature j to program x

\[ M = \{ f, h, \ldots, i, j, \ldots \} \]

Function Composition

- Multi-featured applications are expressions

\[ \text{app1} = i \bullet f \quad \text{app2} = j \bullet h \quad \text{app3} = i \bullet j \bullet f \]

- application with features f and i
- application with features h and j
- your turn...

Given a GenVoca model, we can create a family of applications by composing features

Expression Optimization

- Constants, functions represent both feature and its implementation
  - different functions with different implementations of the same feature

\[ k_1 \bullet x \quad \text{// adds k with implementation #1 to x} \]
\[ k_2 \bullet x \quad \text{// adds k with implementation #2 to x} \]

- When application requires feature \( k \), it is a matter of optimization to determine the best implementation of \( k \)
  - counterpart of relational optimization
  - more complicated rewrites possible too...

- Keys to success of Relational Optimizers
  - represent program designs as expressions
  - rewrite expressions using algebraic identities

- Here's the generalization:

  - domain model is an algebra for a domain or product-line
    - is set of operations (constants, functions) that represent stereo-typical building blocks of programs/members
    - compositions define space of programs that can be synthesized
  - given an algebra:
    - there will always be algebraic identities among operations
    - these identities can be used to optimize expression representations of programs, just like relational optimizers

**AHEAD:** The 2nd Generation

Algebraic Hierarchical Expressions for Application Design

---

**Feature Modularization**

- A feature modularizes multiple refinements, classes
  - ex: R refines class A, interface C, and adds class D

![Diagram of feature modularization]

**Composition**

- Consider constant P and refinement R:
  \[
  P = \{ A_P, B_P, C_P \} \\
  R = \{ A_R, C_R, D_R \}
  \]

- What is \( R \cdot P \)?
Composition

- Align units by name:
  \[
  P = \{ \quad A_P, B_P, \quad C_P \quad \}
  \]
  \[
  R = \{ \quad A_R, \quad C_R, \quad D_R \quad \}
  \]
  \[
  R \odot P = \{ \quad A_R \odot A_P, B_P, \quad C_R \odot C_P, \quad D_R \quad \}
  \]
- Compose units with same name (ignoring subscripts)
- Copy units that aren't refined
- Do the obvious thing...

Law of Composition

\[
R \odot P = \{ \quad A_R, \quad C_R, \quad D_R \quad \} \odot \{ \quad A_P, \quad B_P, \quad C_P \quad \}
\]
\[
= \{ \quad A_R \odot A_P, \quad B_P, \quad C_R \odot C_P, \quad D_R \quad \}
\]
- Fundamental algebraic rewrite of FOP
- Says how composition distributes over modularization
- Do you recognize this law?

Like Inheritance

"class representation"

```java
class P {
    member A_p;
    member B_p;
    member C_p;
}

class R extends P {
    member A_r;
    member C_r;
    member D_r;
}

class R\odot P extends R {}
```

"algebraic representation"

\[
P = \{ \quad A_P, \quad B_P, \quad C_P \quad \}
\]
\[
R = \{ \quad A_R, \quad C_R, \quad D_R \quad \}
\]
\[
R \odot P = \{ \quad A_R \odot A_P, \quad B_P, \quad C_R \odot C_P, \quad D_R \quad \}
\]

Composition Corollaries

- \( f_1, f_2 \) are functions
- \( c_1, c_2 \) are constants
  \[
f_1 \odot f_2 = f_{12} \quad \text{– composite function}
  \]
  \[
c_1 \odot c_2 = c_1 \quad \text{– \( c_1 \) overrides \( c_2 \)}
  \]
  \[
c_1 \odot f_1 = c_1 \quad \text{– \( c_1 \) overrides \( f_1 \)}
  \]
- See examples of these ideas later
Scaling Program Generation

- Generating code for an individual program is OK, but not sufficient
- Today's systems are **not individual programs**, but groups of collaborating programs
  - client-server systems, tool suites (IDEs)
- Further, **systems are not solely defined by code**
  - architects routinely use many knowledge representations
  - formal models, UML models, makefiles, documents, ...

Question

- How does stepwise development scale to the synthesis of multiple programs and multiple-program representations?
- Challenge is not possibility
  - lots of ad hoc ways
  - challenge is to define way that treats all representations – code and non-code – uniformly

Insight #1: Platonic Forms and Languages

- Each program representation captures different information in different languages

Insight #2: Generalize Modularity

- A **module** is a containment hierarchy of related artifacts
- Generalize module hierarchies to arbitrary depth, contents
Modularization of Multiple Programs

- Modules contain all needed representations of a system

Simple Representation

- Module hierarchies = nested sets

Insight #3: Generalize Features

- When a program is refined, any or all of its representations may be updated
- Ex: Add a new feature F to program P changes:
  - code (to implement F)
  - documentation (to document F)
  - makefiles (to build F)
  - formal properties (to characterize F)
  - performance properties (to profile F)
  - ...
- This is a collaboration

#3: Generalize Features

- Containment hierarchy is a “constant”
- Feature is a “function” that maps (transforms) containment hierarchies

- adds new nodes (e.g., new .java, .html files)
- refines existing nodes
**Simple Implementation**

- Feature composition = directory composition
  - produces directory isomorphic to inputs

\[
C = X.java \bullet X.java
\]

**Simple Theory**

- Result computed algebraically by **recursively** expanding and applying the law of composition

\[
C = B \bullet A
\]

\[
= \{ \text{Code}_B, \text{R.drc}_B, \text{Htm}_B \} \bullet \{ \text{Code}_A, \text{R.drc}_A, \text{Htm}_A \}
\]

\[
= \{ \text{Code}_B \bullet \text{Code}_A, \text{R.drc}_B \bullet \text{R.drc}_A, \text{Htm}_B \bullet \text{Htm}_A \}
\]

\[
= \{ \{ X.java_B, Y.java_B \} \bullet \{ X.java_A, Y.java_A \}, \text{R.drc}_B \bullet \text{R.drc}_A, \{ W.htm_B \} \bullet \{ Z.htm_A \} \}
\]

\[
= \{ \{ X.java_B \bullet X.java_A, Y.java_B \bullet Y.java_A \}, \text{R.drc}_B \bullet \text{R.drc}_A, \{ W.htm_B, Z.htm_A \} \}
\]

**Note!**

- Each expression defines an artifact to be produced

\[
C = \{ \{ X.java_B \bullet X.java_A, Y.java_B \bullet Y.java_A \}, \text{R.drc}_B \bullet \text{R.drc}_A, \{ W.htm_B, Z.htm_A \} \}
\]

**Polymorphism...**

- Composition operation \( \bullet \) is **polymorphic**
  - composition law defines how sets are composed
  - different implementation of \( \bullet \) for each representation
    - \( \bullet \) for code
    - another \( \bullet \) for html files, etc.
  - But what does refining a non-code artifact mean?
  - what general principle guides refinement?
Example: Makefiles

- Instructions to build parts of a system
  - it is a language for synthesizing programs
- When we synthesize code for a system, we also have to synthesize a makefile for it
- Sounds good, but...
  - what is a refinement of a makefile?????

Question: what is a general paradigm for refining non-code artifact types?

Makefile

```
make
main
  compile A
  compile B
  compile C
common
  compile X
  compile Y
  compile Z
clean
  delete *.class
```

command line> make main

Makefile Refinements

```
<project myMake>
  <target main depends="common">
    <compile A>
    <compile B>
    <compile C>
  </target>
  <target common>
    <compile X>
    <compile Y>
    <compile Z>
  </target>
  <target>
    <compile D>
    <compile F>
  </target>
  <target>
    <compile E>
    <delete *.ser>
  </target>
  <target>
    <delete *.class>
  </target>
<project>
```

Makefiles Have a Class Structure!

```
class myMake {
  void main {
    <compile A>
    <compile B>
    <compile C>
  }
}
```
Makefile Refinement is Code Refinement

```xml
<project myMake>
  <target main depends="common">
    <compile A/>
    <compile B/>
    <compile C/>
    <compile D>
  </target>
  <target common>
    <compile X/>
    <compile Y/>
    <compile Z/>
    <compile Q>
  </target>
</project>
```

new instructions added after existing instructions

new instructions added after existing instructions

new instructions added after existing instructions

new instructions added after existing instructions

correspondence generalizes to makefile properties such as data members, etc.

correspondence generalizes to makefile properties such as data members, etc.

Big Picture

- Most artifacts today (HTML, XML, etc.) have or can have a hierarchical structure
- But there is no refinement relationship among artifacts!
  - what's missing are refinement operations for artifacts
- Need tools to refine instances of each artifact type
  - MS Word?
    - given such tools, scale stepwise development scales without bounds...
- Modularize changes/additions to all representations of a system
  - so all artifacts (code, makefiles, etc.) are updated consistently
- Compositions yield consistent representations of a system
  - exactly what we want
  - simple, elegant theory behind simple implementation

Insight #4: Principle of Uniformity

- **Principle of Uniformity**
  - create analog in OO representation:
    treat all artifacts equally, as objects or classes
  - refine non-code representations same as code representations
- That is, you can refine any artifact
  - understand it as an object, collection of objects, or classes
- We are creating a structural theory of information based on features
  - it works for code and other representations

Product Member Synthesis Overview

- generalizes RQO paradigm
- scales to large systems

declarative DSL

Engineer

```plaintext
h \circ g \circ f_1
h \circ g_2 \circ f_2
h \circ g_3 \circ f_3
```

artifacts of specified system

generators

```plaintext
\text{generator} \rightarrow \text{artifact}_1
\text{generator} \rightarrow \text{artifact}_2
\text{generator} \rightarrow \text{...}
```
Recommended Readings

- Garlan, Allen, and Ockerbloom, "Architectural Mismatch or Why it is hard to build Systems out of existing parts", ICSE 1995.
- Kiczales and Batory, "Scoping Constructs for Program Generators". Generative and Component-Based Software Engineering (GCSE), September 1999.
- Li, "The Death of Computer Languages, the Birth of Intentional Programming", NATO Science Committee Conference, 1995.
- Ossher and Tarr, "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software.", CACM October 2001.
AHEAD Tool Suite

kick the tires...

Composer Tool

- Key tool in AHEAD Tool Suite (ATS) is composer
- composer expands AHEAD expression to yield target system

```
feat1
feat2
feat3

composer

feat321 = feat3 • feat2 • feat1

> composer -target=feat321 feat1 feat2 feat3
```

Jak Files

- Program in extended-Java files
  - Jak(arta) files
- Java + feature declarations, etc.
  - Jak is an extensible language
- AHEAD is bootstrapped
  - Most AHEAD tools are written in Jak

Other Tools...

- Besides composer
  - `jak2java` – translates Jak files to Java files
  - `javac` – javac compiler
  - `reform` – Jak or Java file formatter/prettier-printer
  - others...

```
> cd <model-directory>
> composer -target=...
> reform *.jak
> jak2java *.jak
> reform *.java
> javac *.java
```
Jak-File Composition Tools

- **composer** invokes Jak-specific tools to compose Jak files
  - two tools now: **jampack** and **mixin**
  - **jak2java** translates Jak to Java

```
class top {
  int a;
  void foo() {...}
}
```

```
refines class top {
  int b;
  int bar() {...}
}
```

#### jampack

- Flattens “inheritance” hierarchies
  - takes expression as input, produces single file as output
  - basically macro expansion with a twist...

```
class top {
  int a;
  void foo() {...}
}
```

```
refines class top {
  int b;
  int bar() {...}
}
```

**jampack** may not be composition tool of choice
- look at typical debugging cycle
- problem: manual propagation of changes
- reason: **jampack** doesn’t preserve feature boundaries

#### mixin

- Encodes class, extensions as inheritance hierarchy

```
abstract class top$$A {
  int a;
  void foo() {...}
}
```

```
public class top extends top$$A {
  int b;
  int bar() {...}
}
```

```
SoUrCe “A/top.jak”
```

```
SoUrCe “B/mid.jak”
```
unmixin

- Edit, debug composed A.jak files
- unmixin propagates changes from composed file to original feature files automatically

Composable Representations

- Current list...
  - *.jak – extended Java files (Jakarta)
    - class
    - interface
    - state machine (ex: embedded DSL)
  - *.equation – named expression files
  - *.b – grammar files
  - *.drc – design rule files
  - others...

AHEAD tools are written in extended Java.
AHEAD has been bootstrapped so that its tools have been written using AHEAD tools.

Demo...

see files, compositions

Cultural Enrichment

- Note algebraic underpinning...
  - Same algebraic paradigm as AHEAD
    - progressively elaborating a containment hierarchy
    - can optimize expression (not this one...)
    - see last lecture!
Cultural Enrichment

- To see connection, watch how containment hierarchy is formed...
  - adding new artifacts is example of module refinement

Big picture: lots of operators on AHEAD modules
- seems that lots of optimizations are possible too... (current work)

A Simple Example

to illustrate concepts, tools

Domain of Graph Applications

- Simple way to express family of related applications is as a grammar
  - different members distinguished by different sets of features

Example Family Members

- choose one
- choose at least one
It is Easy to...

- Imagine a GUI tool that allows you to specify any possible combination
  - declarative language
  - tool generates an explanation of your specification
  - and identifies errors (and suggests corrections) when combinations of features are not possible

See lecture on Verification of Feature Compositions

That’s Easy...

- So too is creating the underlying FOP model:

  \[
  \text{GPL} = \{ \begin{array}{l}
  \text{directed graphs} \\
  \text{undirected graphs} \\
  \text{bfs} \quad \text{-- breadth first search} \\
  \text{dfs} \quad \text{-- depth first search} \\
  \text{cycle checking} \\
  \text{numbering} \quad \text{-- vertex numbering} \\
  \text{regions} \quad \text{-- connected regions} \\
  \end{array} \}
  \]

Constructing Applications

- Graph application: `graph_app = region ● vertex ● dfs ● directed = vertex ● region ● dfs ● directed`

Further Reading

AHEAD Coding Examples
Class and Class Extension Specifications

import initial.stuff;
class myclass {
    int baseVariable;
    // original method is empty
    void baseMethod() {}}

import more.stuff;
refines class myclass {
    // introduce new variable
    int refVariable = 0;
    // introduce new method
    int refMethod() {
        return refVariable;
    }

    void baseMethod() {
        // extension of baseMethod
        // an "execution" around advice in AOP
        int before_stuff = 1;
        Super().baseMethod(); // AOP "proceed"
        int after_stuff = 2;
    }
}

JamPack Composition of Classes in baseRef.equation

layer baseRef;
import initial.stuff;
import more.stuff;
class myclass {
    int baseVariable;
    // introduce new variable
    int refVariable = 0;
    // original method is empty
    final void baseMethod$$base() {}

    void baseMethod() {
        // extension of baseMethod
        // an "execution" around advice in AOP
        int before_stuff = 1;
        baseMethod$$base(); // AOP "proceed"
        int after_stuff = 2;
    }

    // introduce new method
    int refMethod() {
        return refVariable;
    }
}
**Mixin Composition of Classes in baseRef.equation**

```jakbaseRef/myclass.jak```

```jaklayer baseRef;
import initial.stuff;
import more.stuff;
SoUrCe RooT base "../base/myclass.jak";
abstract class myclass$base {
    int baseVariable;
    // original method is empty
    void baseMethod() {}}
SoUrCe ref "../ref/myclass.jak";
class myclass extends myclass$base {
    // introduce new variable
    int refVariable = 0;
    // introduce new method
    int refMethod() {
        return refVariable;
    }
    void baseMethod() {
        // extension of baseMethod
        // an "execution" around advice in AOP
        int before_stuff = 1;
        Super().baseMethod(); // AOP "proceed"
        int after_stuff = 2;
    }
}
```

**AHEAD Coding Examples**

**State Machine and State Machine Extension Specifications**

```jakbase/mysm.jak```

```jakimport something.*;
State_machine mysm {
    Delivery_parameters( Evnt e );
    // start, stop states implicitly defined
    States midpoint;
    Transition begin: start -> midpoint
        condition e != null
        do {
            commonaction( e );
        }
    Transition end: midpoint -> stop
        condition e != null
        do {
            commonaction( e );
        }
    void commonaction( Evnt e ) { /* something */
    }
}
```

```jakref/mysm.jak```

```jakimport evenmore.*;
refines State_machine mysm {
    Transition loop : midpoint -> midpoint
        condition e == null
        do ()
    }
```
JamPack Composition of State Machines in baseRef.equation

layer baseRef;
import something.*;
import evenmore.*;
State_machine mysm {
    Delivery_parameters( Evnt e );
    // start, stop states implicitly defined
    States midpoint;
    Transition begin: start -> midpoint
        condition e != null
        do {
            commonaction( e );
        }
    Transition end: midpoint -> stop
        condition e != null
        do {
            commonaction( e );
        }
    // add new transition
    Transition loop : midpoint -> midpoint
        condition e == null
        do {
            void commonaction( Evnt e ) { /* ... */
        }
}

Mixin Composition of State Machines in baseRef.equation

layer baseRef;
import something.*;
import evenmore.*;
Source Root base "../base/mysm.jak";
abstract State_machine mysm$base {
    Delivery_parameters( Evnt e );
    // start, stop states implicitly defined
    States midpoint;
    Transition begin: start -> midpoint
        condition e != null
        do {
            commonaction( e );
        }
    Transition end: midpoint -> stop
        condition e != null
        do {
            commonaction( e );
        }
    void commonaction( Evnt e ) { /* ... */
    }
}
Source ref "../ref/mysm.jak";
State_machine mysm extends mysm$base {
    // add new transition
    Transition loop : midpoint -> midpoint
        condition e == null
        do {
    }
}
constant layer;
// attributes
extern flowleft Int scale;
extern flowright Bool A;
// preconditions
requires flowleft 4 <= scale;
// postconditions
provides flowright !A;

layer ref;
// attributes
extern flowleft Int scale;
extern flowright Bool B;
// preconditions
requires flowleft scale <= 4;
// postconditions
provides flowright B;

constant layer baseRef;
// externally defined attributes
extern flowright Bool A;
extern flowright Bool B;
extern flowleft Int scale;
provides flowright !A and B;
requires flowleft scale == 4;

// base grammar for mini-calculator
// IDENTIFIER is predefined
// Tokens here

"+" PLUS
// first production is start production

Expr : IDENTIFIER
    | IDENTIFIER Operator Expr :: Opr

Operator : PLUS :: Plus

// adds minus operator
// add new token

"-" MINUS
// import previously defined left-hand side
require Operator;
// add new production

Operator : MINUS :: Minus

// base/grammar.b
// ref/grammar.b
// baseref/grammar.b

// composition of above two files

"-" MINUS
"+" PLUS

Expr : IDENTIFIER
    | IDENTIFIER Operator Expr :: Opr

Operator : MINUS :: Minus
    | PLUS :: Plus

// base/rules.drc
// ref/rules.drc
// composition of above two files
AHEAD Coding Examples
Equations, Equation Extensions, and Composition

# base equation
# = e . d . c (listed in left-to-right order)

```
c
d
e
```

# equation extension
# super references base equation

```
a
b
super
f
g
```

# Generated

```
a
b
c
d
e
f
g
```
Verification of Feature Compositions

correctness of feature expressions

Introduction

- Fundamental problem: not all compositions of features are correct
  - but code can still be generated!
  - and maybe code will still compile!
  - and maybe code will run for a while!
  - impossible for users to figure out what went wrong!

Introduction

- Must verify correctness compositions automatically
  - not all features are compatible
  - selection of a feature may enable others, disable others
- Domain-specific constraints identify legal compositions
- Want process of applying/testing constraints to be automatic
- Presentation overview:
  - tool demonstration
  - present theory behind the tool

Tool Demo

- Illustrate on Graph Product Line
  - has been applied to much larger examples
- Declarative domain specific language
  - domain-specific counterpart to Dell web page
- Propagates constraints among selections
  - cannot write incorrect specification
- Can debug model specifications
  - by testing if expected compositions are indeed compatible or incompatible
Feature Diagrams and Grammars
(The Theory Behind The Scenes)

Feature diagrams are standard product-line notations:
- declarative way to specify products by selecting features
- FDs are trees:
  - leaves are primitive features
  - internal nodes are compound features
  - parent-child are containment relationships

Feature Diagrams
- Mandatory – features that are required
- Optional – features that are optional
- And – all subfeatures (children) are selected
- Alternative – choose only 1 subfeature
- Or – 1+ or 0+ subfeatures can be selected
**Example**

What is a legal product specification?

- E is?
- R is?
- S is?

Sound familiar?
- de Jonge and Visser 2002: *FDs are graphical representations of grammars*
- “GenVoca Grammars” 1992: *grammar defines legal orders in which features can be composed*

**Graph Product Line (GPL)**

\[
Gpl = \{
\begin{align*}
& \text{DIRECTED} & \text{- directed graphs} \\
& \text{UNDIRECTED} & \text{- undirected graphs} \\
& \text{BFS} & \text{- breadth first search} \\
& \text{DFS} & \text{- depth first search} \\
& \text{CYCLE} & \text{- cycle checking} \\
& \text{NUMBER} & \text{- vertex numbering} \\
& \text{REGIONS} & \text{- connected regions} \\
& \ldots &
\end{align*}
\}
\]

**GPL Grammar**

\[
Gpl : \text{Alg}^+ \left[ \text{Src}, \text{Wgt} \right] \text{Gtp};
\]

- Gtp : DIRECTED | UNDIRECTED;
- Wgt : WEIGHTED | UNWEIGHTED;
- Src : DFS | BFS;
- Alg : NUMBER | CONNECTED | STRONGC | CYCLE | MSTPRIM | MSTKRUSKAL | SHORTEST;

A sentence of this grammar defines a composition of features

\[
\text{Prog} = \text{NUMBER} \cdot \text{CYCLE} \cdot \text{BFS} \cdot \text{UNWEIGHTED} \cdot \text{DIRECTED}
\]

**Mapping of FDs to Grammars**

\[
S : e_1 \left[ e_2 \right] \cdot \ldots \cdot \cdot \end{align*}
\]

\[
S : e_1 \mid e_2 \mid \ldots \cdot \end{align*}
\]

\[
S : e_1 \mid e_2 \mid \ldots \cdot \end{align*}
\]

... \[
S^+ \cdot \end{align*}
\]

\[
S : e_1 \mid e_2 \mid \ldots \cdot \end{align*}
\]
Example: Convert FD to Grammar

Grammar

Application defined by Feature Model = sentence of grammar E
Resulting grammar is a GenVoca grammar (1992)

Propositional Formula

Set of boolean variables and propositional logic predicate that constrains values of these variables

Standard $\neg$, $\lor$, $\land$, $\Rightarrow$, $\Leftrightarrow$ operations

Nonstandard:
- atmost1($e_1...e_k$) – at most one $e_i$ is true
- ...

Insight: A grammar is a compact representation of a propositional formula

Mapping Productions to Formulas

Given production: $R : P_1 | ... | P_n$;

$R$ can be referenced in two ways:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>... $R+$ ... (choose 1 or more)</td>
<td>$R \Leftrightarrow P_1 \lor P_2 \lor ... \lor P_n$</td>
</tr>
<tr>
<td>... $R$ ... (choose 1)</td>
<td>$R \Leftrightarrow (P_1 \lor P_2 \lor ... \lor P_n) \land \text{atmost1}(P_1,P_2,\ldots,P_n)$</td>
</tr>
</tbody>
</table>
Mapping Patterns to Formulas

- \( T_1 \ T_2 \ldots \ T_n :: P \)
  - formula: \( P \iff T_1 \land P \iff T_2 \land \ldots \land P \iff T_n \)

- \( T_1 \ [ T_2 ] \ldots \ T_n :: Q \)
  - formula: \( Q \iff T_1 \land T_2 \iff Q \land \ldots \land Q \iff T_n \)

Example: Grammars to Formulas

- Convert each production, pattern to formula
- Take conjunction of all formulas
- Conjoin root of grammar

Grammar:
\[
E : R \ S ;
R : g | h | i ;
S : a [ b ] c ;
\]

Propositional formula:
\[
E \iff R \land E \iff S \\
R \iff (g \lor h \lor i) \land \text{atmost1}(g, h, i) \\
S \iff a \lor b \iff S \iff c \\
E
\]

A sentence of E satisfies the propositional formula and vice versa

Last Example

Recap

- We can map any AHEAD model or Feature Diagram to a propositional formula
- But what about constraints?
  - Any additional, arbitrary propositional formulas conjoined onto grammar formula
    - Ex: if features i and b are incompatible, we would add the formula
      \[ i \lor b \Rightarrow \neg (b \land i) \]
Example: Additional Constraints in GPL

- Straight from Graph Algorithm Text

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Required Graph Type</th>
<th>Required Weight</th>
<th>Required Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Numbering</td>
<td>Any</td>
<td>Any</td>
<td>BFS, DFS</td>
</tr>
<tr>
<td>Connected Components</td>
<td>UNDIRECTED</td>
<td>Any</td>
<td>BFS, DFS</td>
</tr>
<tr>
<td>Strongly Connected Components</td>
<td>DIRECTED</td>
<td>Any</td>
<td>DFS</td>
</tr>
<tr>
<td>Cycle Checking</td>
<td>Any</td>
<td>Any</td>
<td>DFS</td>
</tr>
<tr>
<td>Minimum Spanning Tree</td>
<td>UNDIRECTED</td>
<td>WEIGHTED</td>
<td>None</td>
</tr>
<tr>
<td>Single-Source Shortest Path</td>
<td>DIRECTED</td>
<td>WEIGHTED</td>
<td>None</td>
</tr>
</tbody>
</table>

GPL Model Specification

```
Gpl : Alg* [Src] Wgt Gtp;
Gtp : DIRECTED | UNDIRECTED ;
Wgt : WEIGHTED | UNWEIGHTED ;
Src : DFS | BFS ;
Alg : NUMBER | CONNECTED |
     | STRONGC | CYCLE | MSTPRIM |
     | MSTKRUSKAL | SHORTEST ;
%%
NUMBER implies Gtp and Src;
CONNECTED implies UNDIRECTED and Src;
CYCLE implies Gtp and DFS;
SHORTEST implies DIRECTED and WEIGHTED;
STRONGC implies DIRECTED and DFS;
MSTKRUSKAL or MSTPRIM implies UNDIRECTED and WEIGHTED;
```

Recap

- An AHEAD Domain Model is a propositional formula!
  - primitive features and compound features are variables
- Grammar:
  - specifies order in which features are composed
  - ordering very important for AHEAD
- Additional propositional constraints:
  - weed out incompatible feature combinations

Declarative Domain-Specific Languages

GenVoca Grammar  ➔ Feature Diagram  ➔ Propositional Formula  ➔ DDSLs
Declarative Languages

- Features enable declarative program specifications
  - that’s what feature diagrams are for!
  - counterpart of SQL, Dell web pages

- Want a declarative GUI DSL that acts like a syntax-directed editor
  - user selects desired features
  - tool precludes specifying incorrect programs

Constraint Propagation

- 1980’s result from Artificial Intelligence

- Logic Truth Maintenance System
  - boolean constraint propagation (BCP) algorithm
  - takes a boolean predicate, set of variable assignments as input, deduces other variable assignments as output
  - very simple, efficient algorithm


Debugging Feature/Domain Models

- We know features A and B are compatible
  - let $P_{\text{model}}$ be the predicate of our feature model
  - $P_{\text{model}} \land A \land B$ must be satisfiable
    - that is, there is a product that has both A and B

- Satisfiability (SAT) Solver
  - off-the-shelf tool that automatically determines if a boolean predicate is satisfiable
  - very efficient

- Basis for feature model debugging
  - provide a script of compatible, incompatible features and verifies these conditions hold
  - solver confirms known properties of a model
Experience

- Have worked well...
- Predicates are simple
- Use off the shelf constraint solvers
- Reason: architects think in terms of features
  - if predicates were really complicated
    - architects couldn’t design
    - people couldn’t program
    - because it would be too difficult
- We are making explicit what is implicit now...

There’s More...

- Benavides noticed you could add numerical attributes to grammar

There’s More... and is Really Exciting

- Allow features to have additional parameters
  - property lists
- Generalize predicates to include constraints on numeric variables
  - restrict models based on feature requirements, criteria
  - select product that maximizes/minimizes criteria (performance!)
  - use standard Constraint Satisfaction Problem (CSP) Solvers
- Automatically find optimal system configurations
  - true automatic programming!
  - counterpart to relational query optimizers

Recommended Readings

Feature Oriented Model Driven Development: A Case Study for Portlets

work with:
S. Trujillo and O. Diaz
University of Basque Country
San Sebastian, Spain

Introduction

- **Model Driven Development (MDD)** is an emerging paradigm for software creation
  - uses **domain-specific languages (DSL)**
  - encourages automation
  - exploits data exchange standards

- **Model is written in a DSL**
  - captures particular details of program's design
  - several models are needed for a program
  - models can be derived by transformations
  - program synthesis is transforming high-level models into executables (which are also models)

Feature Oriented Model Driven Design (FOMDD) is blend of FOP and MDD

- models can be refined by composing features
  - **endogenous transformations** that map models expressed in the same DSL

- models can be derived from other models
  - **exogenous transformations** that map models written in different DSLs

This Lecture: Case Study of FOMDD

- Product line of **portlet**s – building blocks of web portals

- Explain how a portlet is specified as a set of models from which we refine and derive an implementation

- Work exposes a fundamental relationship between model derivation and model refinement
  - a transformation of a composition of models = a composition of transformed models

- Practical: helped validate the correctness of our abstractions, tools, and portlet specifications, as well as optimizing portlet synthesis
General Results

- #1: Suggest a simple way to understand and appreciate MDD (beyond the hype)
  - FOP and MDD are examples of metaprogramming

- #2: Insight into a theory behind FOMDD
  - there is a science behind engineering

- Start with review of portlets & MDD

Background: Portlets

What is a Portal?

- Web page that provides centralized access to a variety of services

- Services are often offered by a 3rd party component called a portlet

Portlet Architecture

- 3-tier architecture
  - end-user’s MyBrowser accesses MyPortal page that is hosted by Consumer
  - MyPortal layout aggregates Alpha, Beta, Delta portlets offered by different producers
Portlets

- Offer not only business logic methods
- Additionally provide presentation-oriented interface
- Return not only raw data, but renderable markup (XHTML)

- Portlet development simplified with release of:
  - Web Services for Remote Portlets (WSRP)
  - Java Specification Request 168 (JSR 168)
  - how to implement portlets in Java
  - fostered a COTS market where portlets can be deployed independently of the platform on which they were developed
  - different customers demand different portlets that overlap in functionality

- Consequence: production techniques for portlets increasingly sought

Portlets Continued

- Experience: different portlets share sizable fraction of code (50-60%)

- Created OO framework to encapsulate and reuse common logic
  - 85 classes, 9K Java LOC

- Created a Domain Specific Language for Portlets (PSL) to define portlet functionality
  - PSL specification is represented by multiple XML docs
  - can synthesize portlet implementation

Background: Model Driven Development

MDD Overview

- Program development based on writing of one or more models to specify a target program
  - ideally, models are platform independent (PIM)
  - model transformations convert PIMs into platform specific models (PSMs), by introducing assorted details (e.g., technology bindings)
  - transformations can produce input to an analysis tool, or executable (both of which are models)
  - Bezivin “Everything is a Model”
**MDD Bandwagon**

- OMG's **Model Driven Architecture (MDA)**
  - define models in terms of UML
  - transform models using graph transformations (QVT)
- First (and best) work I've seen is Vanderbilt's **Model Integrated Computing (MIC)**
- Microsoft recent entry: Software Factories
- Lots of others...
  
  [http://www.modelbased.net/mda_tools.html](http://www.modelbased.net/mda_tools.html)

**Metaprogramming**

- Concept that program development is a computation
  - my claim: MDD is a metaprogramming paradigm
  - models are values
  - transformations are functions that map values
- MDD scripts are **metaprograms**
  - programs that manipulate values that themselves are programs

**Familiar Example**

- **ant** makefiles are metaprograms
  - values of a makefile are files (programs)
  - execution of a makefile produces an executable
- MDD process is a makefile (metaprogram) whose inputs are DSL specifications of target programs (models) and whose output values are synthesis targets

**Metaprogramming and FOP**

- **FOP** is a metaprogramming paradigm
  - programs are values
  - features are functions that map values
  - program synthesis is (meta) expression evaluation
- **FOP and MDD** are complementary paradigms
- Now look at an example...
The PinkCreek Case Study

Model Driven Development of Portlets

PinkCreek

- Family of portlets that provide flight reservation capabilities
  - search for flights
  - select flight options
  - reserve flights
  - purchase tickets

Step 1: Define Portlet Controller

- As a state machine
- Use State Charts
  - platform independent model representing flow of computations in a portlet
  - use W3C SCXML language, tools for specifications

Step 2: Map SC to PSL

- Portlet Specification Language (PSL)
- PSL spec consists of 3 XML documents
  - PSL$_{ctrl}$ – state machine of controller
  - PSL$_{act}$ – actions to be performed
  - PSL$_{view}$ – views to be rendered
- Interpreter runs controller, invokes actions and views during portlet execution
Step 2.1: Map SC to PSL

- PSL has its own encoding of state machines
  - created before SCXML standard defined
  - function $T_{sc2ctrl}$ translates XML documents

$T_{sc2ctrl} \rightarrow \text{PSL}_\text{ctrl} = T_{sc2ctrl}(\text{SC})$

Step 2.2: PSL$\_\text{ctrl}$ to PSL$\_\text{act-sk}$

- Inside PSL$\_\text{ctrl}$, names of view and model actions are listed
- Harvest names of these actions into PSL$\_\text{act-sk}$

```xml
<scxml version="1.0" initialstate="s1Search">
  <state id="s1Search">
    <transition event="processAction">
      <target next="s1Search"/>
      <action>LoadAirport</action>
    </transition>
    <transition event="doDispatch">
      <target next="s2Select">
        <action>SearchFlightView</action>
      </transition>
    </state>
  </scxml>
```

$\text{PSL}_\text{ctrl} = T_{sc2ctrl}(\text{SC});$

$\text{PSL}_\text{act-sk} = T_{cntl2act}(\text{PSL}_\text{ctrl});$

$\text{Tcntl2act}$

Step 2.3: PSL$\_\text{act-sk}$ to PSL$\_\text{act}$

- Add parameter types by composing hand-written refinement
- Producing PSL$\_\text{act}$ document

```xml
<Portlet.actions id="PinkCreek">
  <Action id="LoadAirport" type="JAVA_CLASS">
    <class>pinkcreek.LoadAirport</class>
    <!-- TODO Params --><!-- TODO Results -->
  </Action>
</Portlet.actions>
```

$\text{PSL}_\text{ctrl} = T_{sc2ctrl}(\text{SC});$

$\text{PSL}_\text{act-sk} = T_{cntl2act}(\text{PSL}_\text{ctrl});$

$\text{PSL}_\text{act} = \Delta \text{PSL}_\text{act-sk} \cdot \text{PSL}_\text{act-sk};$

// same is done with view actions

Where We Are...

PSL framework can run the portlet given the above documents

What’s missing are:
1. business logic for each action,
2. logic to draw each view

PSL$\_\text{ctrl} \rightarrow \text{PSL}_\text{ctrl};$

PSL$\_\text{act-sk} = T_{cntl2act}(\text{PSL}_\text{ctrl});$

PSL$\_\text{act} = \Delta \text{PSL}_\text{act-sk} \cdot \text{PSL}_\text{act-sk};$

// same is done with view actions
Step 3: PSL to Implementation

- Big picture:
  - transform PSL$_{act}$ to code
  - transform PSL$_{view}$ to JSP code
  - package in WAR (web archive file)

Step 3.1: Create Code Skeleton

- Translate each PSL$_{act}$ action into Java class + other members and methods needed by framework

```java
import java.util.Hashtable;
import org.onekin.pf.action.jc.IJavaAction;
public class LoadAirport implements IJavaAction {
    /** This is the default Constructor ... */
    public LoadAirport(){ /* empty */ }
    /** This method executes ... */
    public void execute(Hashtable prm, Hashtable rs) { /* empty */ }
}
```

Step 3.2: Add Business Logic

- By composing hand-written refinement
- Produces code module

```java
import java.util.Hashtable;
import org.onekin.pf.action.jc.IJavaAction;
public class LoadAirport implements IJavaAction {
    /** This is the default Constructor ... */
    public LoadAirport(){ /* business logic added */ }
    /** This method executes ... */
    public void execute(Hashtable prm, Hashtable rs) { /* business logic added */ }
}
```

Portlet Synthesis Metaprogram

```
T$_{mkraw}$ produces the “raw material” of a portlet
```
Portlet Synthesis Metaprogram

- \( T_{\text{raw2war}} \) converts raw material into a Web Archive (WAR) file

```
Ttmraw2war(SC, ΔPSL_{act-usr}, ΔPSL_{user}, ΔCode_{usr}, ΔJSP_{usr})
{
  SC = Ttmraw2war(SC);
  ΔPSL_{act-usr} = ΔPSL_{act-usr} * SC;
  ΔPSL_{user} = ΔPSL_{user} * SC;
  ΔCode_{usr} = ΔCode_{usr} * SC;
  ΔJSP_{usr} = ΔJSP_{usr} * SC;
  return [SC, ΔPSL_{act-usr}, ΔPSL_{user}, ΔCode_{usr}, ΔJSP_{usr}];
}
```

Experience

- \( T_{\text{tmraw}} \) automates significant and tedious tasks in portlet development
  - automates the production of boilerplate code
  - engineers only provide the parts that are specific to a desired portlet

- Typical example:
  - from an input of 10 files with 730 LOC, 59 files with 4250 LOC are derived

Big Picture

- Input to \( T_{\text{tmraw}} \) is a 5-tuple

\(<\text{SC, ΔPSL}_{\text{act-usr}}, \text{ΔCode}_{\text{usr}}, \text{ΔPSL}_{\text{user}}, \text{ΔJSP}_{\text{usr}}>\)

<table>
<thead>
<tr>
<th>state machine</th>
<th>action signature</th>
<th>action business logic</th>
<th>view signature</th>
<th>view definition</th>
</tr>
</thead>
</table>

- or \(<s, a, u, v, j>\)

\( \text{Portlet}_{\text{raw}} = T_{\text{tmraw}}(<s, a, u, v, j>) \)

And Now, Features
Need for Customizing Portlets

- Portlets are like other applications
  - family of related portlet designs and capabilities will arise
  - different designs are differentiated in terms of features
  - hundreds of different possibilities

- Lots of features in PinkCreek portlets (26 so far):
  - core -- base portlet
  - assistance -- airport accessibility needs
  - checkin -- check in
  - banner -- commercial banner
  - meal -- meal selection
  - direct -- only direct flights listed
  - refundable -- allows flights to be refundable
  - seat -- seat assignments
  - weather -- check weather at destination
  ...

Portlet 5-Tuple Synthesis

\[ P = F_2 \bullet F_1 \bullet C \]
\[ = \langle \Delta s_2, \Delta a_2, \Delta c_2, \Delta v_2, \Delta j_2 \rangle \]
\[ = \langle \Delta s_1, \Delta a_1, \Delta c_1, \Delta v_1, \Delta j_1 \rangle \]
\[ = \langle s, a, c, v, j \rangle \]
\[ \text{Portlet}_{\text{raw}} = T_{\text{mkr}}(P) \]

Refinements

- State Chart Refinements
  - add new states, transitions, actions, views

[State Chart Diagram]

\[ \Delta S_{\text{sc}} \bullet \text{Base}_{\text{sc}} \]
\[ s_{\text{View}} \]
Remaining Terms of $\Delta$-Tuple

- Metaprogram, similar to that shown earlier

- $T'_\text{mkraw}(...)$ produces a $\Delta F_{\text{raw}}$ – the changes a $\Delta$-tuple makes to a portlet raw file

Portlet Synthesis!

- Portlet raw files are synthesized by composing raw materials of each 5-tuple

  \[
  P = F_2 \cdot F_1 \cdot C
  \]

What $T'_\text{mkraw}$ Does Visually

Experience

- $T'_\text{mkraw}$ automates significant and tedious tasks in portlet feature development
  - automates the production of boilerplate code
  - just specify the parts particular to a portlet feature

- Seat feature in PinkCreek:
  - from an input of 9 files with 163 LOC
  - 27 files with 755 LOC are derived
But Wait...

This isn’t the way we originally intended to synthesize portlets!

Recall

- Original goal: synthesize 5-tuple and then transform
  
  \[ <>\text{portlet} = <>\text{checkin} \cdot <>\text{seat} \cdot <>\text{base} \]
  
  \[ \text{Portlet}_{\text{raw}} = T_{\text{mkraw}}( <>\text{portlet} ) \]

- But instead composing raw files of features:
  
  \[ \text{Portlet}_{\text{raw}} = T'_{\text{mkraw}}( <>\text{checkin} ) \cdot T'_{\text{mkraw}}( <>\text{seat} ) \cdot T_{\text{mkraw}}( <>\text{base} ) \]

Commutativity Diagram

- Commutativity property of our method of synthesizing portlets

\[
T_{\text{mkraw}}( <>\text{F}_1 \cdot <>\text{base} ) = T'_{\text{mkraw}}( <>\text{F}_1 ) \cdot T_{\text{mkraw}}( <>\text{base} )
\]

Fundamental Idea

- Relationship between model derivation and model refinement

\[
f( \Delta M \cdot M_0 ) = ff( \Delta M ) \cdot f(M_0)
\]

the transformation of a composition of tuples equals the composition of the transformation of each tuple
Lots of Commutativity Diagrams

\[ T_{\text{sc2ctrl}}(\Delta CT \cdot SC) = T'_{\text{sc2ctrl}}(\Delta CT) \cdot T_{\text{sc2ctrl}}(SC) \]

Insight

- Reason why commutativity works
  - Structure preserving transformations
  - Proving this formally is difficult
    - Formalizations of input, output domains;
    - Formalization of properties to be preserved;
    - Faithful implementation of formalization

Insight

- Validate by computing results both ways (paths A and B), and compare
  - DIFF using source equivalence (SE)
    - SE is syntactic equivalence with two relaxations:
      - Allows permutations of members when order is not important
      - White space can differ when not important

Experience

- Our tools initially didn’t satisfy commutativity properties
  - Synthesizing via paths A and B yielded different results
  - Discovered errors in our tools & specifications

- Significance of commutativity properties
  - Validation checks provides assurances on the correctness of our PMDD abstractions, portlet specifications, and our tools
  - Applies also to individual transformations (as they too have commutativity diagrams)
  - Big win
Property of Commutativity Diagrams

- Diagrams can be “pasted” together
- Given object in upper left, often want to compute object in lower right
- Any path from upper left to lower right should produce same result

PinkCreek Commutativity Diagrams

- Configuration space of artifacts derived from state chart
  - $T_{\text{mkraw}}$
- Mesh of commutativity diagrams are swept as features are composed
  - $T_{\text{mkraw}}$

Portlet Synthesis

- Start at upper left compute nodes on lower right
  - any path will compute same result – but some paths will be faster!
- #1: refine models and then derive
- #2: derive representations and then refine
- #2 is faster than #1 by factor of 2-3 times

Benefit: Interesting Optimization

- Which way is faster?
  - (A) compose transformations
  - (B) transform compositions
So What?

What can others learn from this work?

Perspective

- MDD is an emerging discipline
  - immature
  - very little agreement on terms, notations
  - confusion will take years to fix

- Don’t need this, and it is unnecessary
  - idea of *metaprogramming*, functional languages and functional programs are useful to express synthesis scripts
  - ideas are latent in MDD now

Theory to Support FOMDD

- If you don’t like functional languages, can use OO languages instead
  - object = program
  - method = transformation
  - class = metamodel = product line of programs

- ideas of MDD models and scripts become simple to understand


Category Theory
#1: A Category

- Directed graph with special properties
- Nodes are **objects**, edges are **arrows**
- Arrows are functions
- Arrows compose like functions, composition is associative
- Identity arrows

Space of Derived Artifacts in PinkCreek

- is a category!
- Objects are artifacts
- Arrows are T transformations

Space of Derived Artifacts

metaprogram T\_mkraw for producing a portlet from a tuple

```cpp
Tmkraw(sc, ctrl, act-sk, view-sk, code-sk, jsp-sk) {
    if (ctrl == ctrl0) {
        act-sk = Tskview (Tskctrl (act-sk));
        view-sk = Tview-sk (Tview-sk (view-sk));
        code-sk = Tcode-sk (Tcode-sk (code-sk));
        jsp-sk = Tjsp-sk (Tjsp-sk (jsp-sk));
        return Scsk (sc, ctrl, act-sk, view-sk, code-sk, jsp-sk);
    }
}
```

#2: Functor

- is a mapping from one category to another

```cpp
Tmkraw(sc0, ctrl0, act-sk0, view-sk0, code-sk0, jsp-sk0) {
    ... (similar to Tmkraw above)
    return Scsk (sc1, ctrl1, act-sk1, view-sk1, code-sk1, jsp-sk1);
}
```
Functor

- is a mapping from one category to another

Features Are Functors!

- Features map space of artifacts by refining them
- Sequentially map one category to another
- Sweeps the commutativity diagrams that we have to traverse to synthesize portlet raw materials

There’s a LOT More

- Category Theory unifies a lot of results in FOP and MDD
  - surprisingly simple and elegant
  - sketches next generation paradigm for FOMDD, tools...
- Stay tuned....

Conclusions
Conclusions

- FOP and MDD are complementary paradigms
  - MDD derives models
  - FOP refines models

- Unified by **metaprogramming**
  - models are values
  - transformations and compositions are functions

- Combined yields a powerful paradigm
  - MDD for product lines

Presented a case study of FOMDD
  - product line of portlets

Exposed a fundamental relationship between model refinement and model derivation
  - commutativity diagrams
    - transformation of composition = composition of transformations
  - allowed us to validate domain abstractions, specifications, and tools
  - optimize portlet synthesis (if validation is not an issue)

Lays groundwork for a theory of FOMDD
  - based on elementary ideas from category theory

Although this was just a case study...

There's nothing domain-specific about the need for MDD, FOP, and their benefits
  - ideas can be applied to many domains

Portlets are an example of many domains where both technologies produce a result that is better than either could deliver in isolation

Fresh perspective where mathematical properties – in addition to engineering feats – guide a principled design of systems

First of many results using advanced software develop techniques

Recommended Readings

- Bezivin, "Model Driven Engineering: Principles, Scope, Deployment, and Applicability", in Laemmel (below).