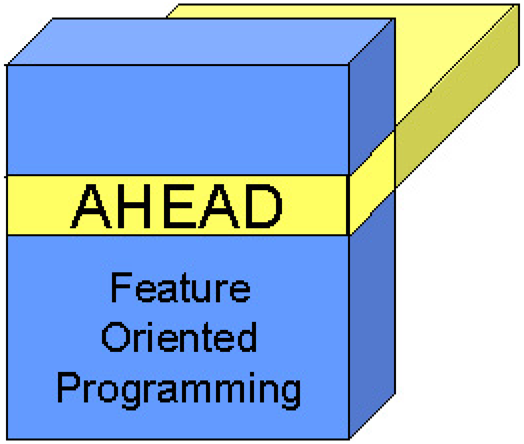
Feature Modularity in Software Product-Lines

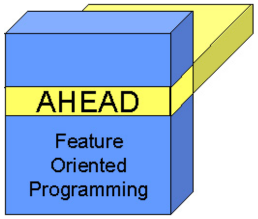
Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy



Feature Modularity in Software Product-Lines

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



Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy

Introduction

- A **product-line** is a family of similar systems
  - Chrysler mini-vans, Motorola radios, software
- Motivation: economics
  - amortize cost of building variants of program
  - design for family of systems
- Key idea of product-lines
  - members 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

Introduction

- **Feature Oriented Programming (FOP)** is the study of feature modularity in product-lines
  - features are first-class entities in design
  - often implemented by collaborations
- History of applications
  - 1986 database systems
  - 1989 network protocols
  - 1993 data structures
  - 1994 avionics
  - 1997 extensible compilers
  - 1998 radio ergonomics
  - 2000 prog. verification tools
  - 2002 fire support simulator
  - 2003 AHEAD tool suite
  - 2004 robotics controllers
  - 2006 peer-to-peer networks

Very Rich Technical Area...

- Integrates many subjects:
  - compilers
  - grammars
  - artificial intelligence
  - databases
  - algebra
  - category theory
  - programming languages
  - compositional programming
  - compositional reasoning
  - OO software design
  - metaprogramming
  - domain-specific languages
  - declarative languages
  - tensors
  - generative programming
  - model driven design
  - verification
  - collaborations
  - refactoring
  - automatic programming
  - aspect-oriented programming

others...

Overall Goal

- Place automation of large-scale software design and construction on a practical and firm mathematical foundation
- Feature orientation allows us to do this in a simple way
- Tutorial shows how...

Tutorial Overview

buckle up!

Lecture 1:	Introduction to FOP
Lecture 2a:	Tool Demos
Lecture 2b:	Verification of Feature Compositions
Lecture 3:	Program Refactoring, Synthesis, and Model-Driven Design
Lecture 4:	Feature Interactions and Program Cubes

Introduction to FOP

a general approach to product synthesis

Motivation

- Software products are:
  - increasing in complexity
  - increasing in costs to develop and maintain
  - decreasing in ability to understand
- Goal of SE is to manage and control complexity
  - structured programming to
  - object-oriented programming to
  - component-based programming to...

  progressively increasing abstractions

  - **today's design techniques are too low-level, expose 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

---

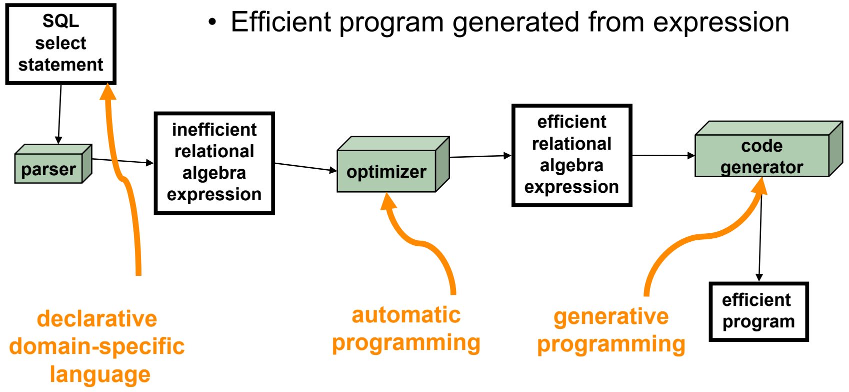
Not Wishful Thinking...

- Example of this futuristic paradigm realized 30 years ago
  - around time when many AI researchers gave up on automatic programming

Relational Query Optimization

Relational Query Optimization

- Declarative query is mapped to an expression
- Each expression represents a unique program
- Expression is optimized using rewrite rules
- Efficient program generated from expression



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 a domain
  - relational algebra
- Represented program designs as **expressions**
  - compositions of relational operations
- Defined 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
- 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

---

Towards a Science of Software Design

What motivates FOP and
how is it defined?

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 and reason about systems

---

A Thought Experiment...

- Look at how people describe programs now...
  - don’t say which DLLs are used...
- Instead, say what **features** a program offers its clients

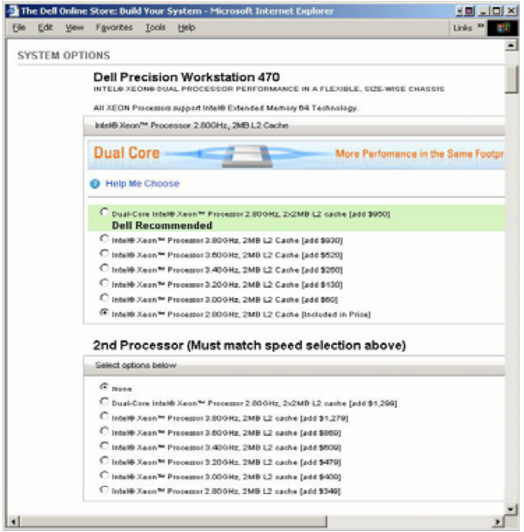
Program1 = feature_X + feature_Y + feature_Z

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**



SYSTEM OPTIONS

Dell Precision Workstation 470

INTEL® XEON® DUAL PROCESSOR PERFORMANCE IN A FLEXIBLE, SIZE-WISE CHASSIS

All XEON Processors support Intel® Extended Memory 64 Technology.

Intel® Xeon™ Processor 2.80GHz, 2MB L2 Cache

Dual Core More Perfomance in the Same Footpr

Help Me Choose

○ Dual-Core Intel® Xeon™ Processor 2.80GHz, 2x2MB L2 cache [add $950]
Dell Recommended
○ Intel® Xeon™ Processor 3.80GHz, 2MB L2 Cache [add $930]
○ Intel® Xeon™ Processor 3.60GHz, 2MB L2 Cache [add $520]
○ Intel® Xeon™ Processor 3.40GHz, 2MB L2 Cache [add $260]
○ Intel® Xeon™ Processor 3.20GHz, 2MB L2 Cache [add $130]
○ Intel® Xeon™ Processor 3.00GHz, 2MB L2 Cache [add $60]
● Intel® Xeon™ Processor 2.80GHz, 2MB L2 Cache [Included in Price]

2nd Processor (Must match speed selection above)

Select options below

● None
○ Dual-Core Intel® Xeon™ Processor 2.80GHz, 2x2MB L2 cache [add $1,299]
○ Intel® Xeon™ Processor 3.80GHz, 2MB L2 cache [add $1,279]
○ Intel® Xeon™ Processor 3.60GHz, 2MB L2 cache [add $869]
○ Intel® Xeon™ Processor 3.40GHz, 2MB L2 cache [add $609]
○ Intel® Xeon™ Processor 3.20GHz, 2MB L2 cache [add $479]
○ Intel® Xeon™ Processor 3.00GHz, 2MB L2 cache [add $400]
○ Intel® Xeon™ Processor 2.80GHz, 2MB L2 cache [add $349]

Chinese Menu – Declarative DSL

Methodology for Construction

- What methodology builds systems by progressively adding details?

- **Step-Wise 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

What is a Feature?

- **Feature**
  - an elaboration or augmentation of an entity(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 Interchangeable

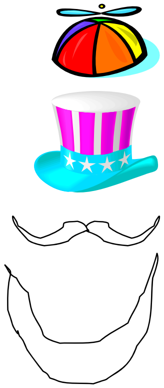
Features are Interchangeable

Features are Interchangeable

Features are Interchangeable

Features are Reusable

Features are Functions!



PersonPhoto beanie(PersonPhoto x)

PersonPhoto uncleSam(PersonPhoto x)

PersonPhoto mustache(PersonPhoto x)

PersonPhoto lincolnBeard(PersonPhoto x)

Composing Features

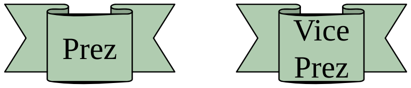
- Feature composition = function composition

= lincolnBeard(uncleSam(

))

Large Scale Features

- Called **Collaborations (1992)**
  - simultaneously modify multiple objects/entities
  - refinement of single entity is called **role**
- Example: Positions in US Government
  - each defines a role



....

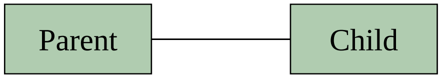
Composing Collaborations

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

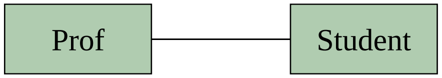
Example of dynamic composition of collaborations

Other Collaborations

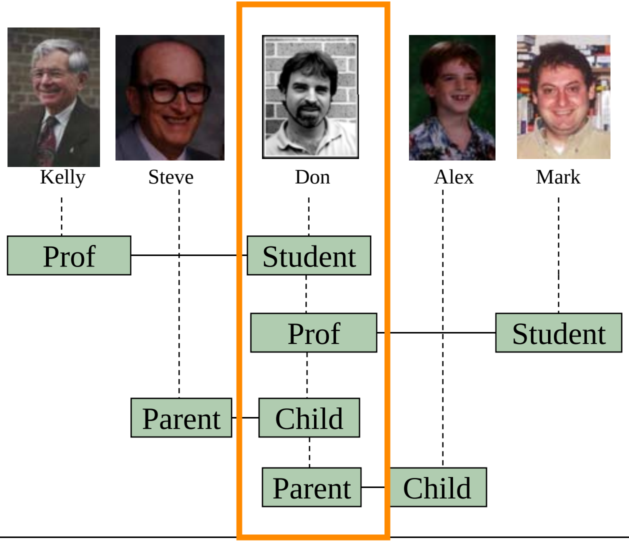
- Parent-Child collaboration



- Professor-Student collaboration



Example



Same Holds for Software!

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

---

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 step-wise development
  - advocates complex programs constructed from simple programs by incrementally adding features
- How are features and their compositions modeled?

The Theory

GenVoca and AHEAD

A Clue...

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

```
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 C {
   member m1;
   member m2;
   member m3;
   member m4;
}
```

=

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

=

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

class C3 extends C2
   member m3;
}

class C23 extends C3 {}
```

Observe…

- Significance: 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, step-wise 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 ]
```

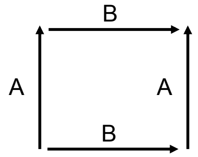
A Closer Analogy

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

```
C1 = (m1,0,0,0)
C2 = (0,m2,0,0)
C3 = (0,0,m3,0)
C4 = (0,0,0,m4)
```

C1 → C2 → C3 → C4 ≠ C4 → C3 → C2 → C1

A → B ≠ B → A



path followed by
A → B is different
than B → A;
end point is the same

Operation We Want...

- Is not quite inheritance...
  - want to add new methods, new fields, and refine existing methods like inheritance
  - also want constructors to be inherited and refined as well, (inheritance doesn't provide this)

```
class C1 {
   constructor1
}
```

●

```
class C2 {
   constructor2
}
```

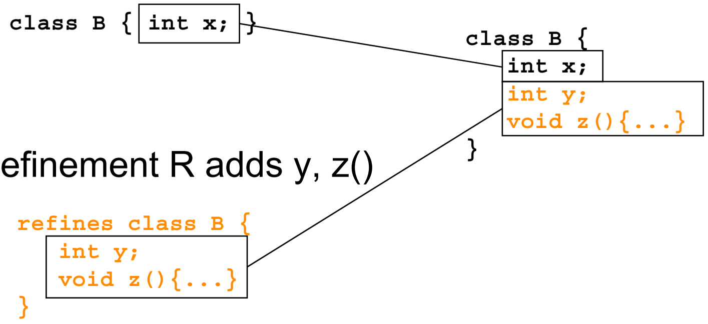
=

```
class C12 {
   constructor1
   constructor2
}
```

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:
- Refinement R adds y, z()



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

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

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

Algebraic Formulation

- Base programs are **constants**

```
// constant P

class B { int x; }
```

- Refinements are **functions**

```
// function R

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

- Composition is an **expression**

```
N = R( P )

  = R ● P
```

- yields:

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

Treat programs as values is metaprogramming

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 ) ) )

    = C4 • C3 • C2 • C1
```

Note:
both notations
are equivalent

Method Refinement ala Inheritance

```
result =    method_refinement      •   base_method

        void foo() {
           /* before stuff */      void foo() {
   =       super.foo();        •      /* do something */
           /* after stuff */       }
        }

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

is substitution or an equivalent encoding

Connecting the Dots...

- Scalability
  - refinement is not limited to a single class
  - **collaborations** modularize refinements of multiple classes **and add new classes**
    - » adding new classes that can be refined is **critical**

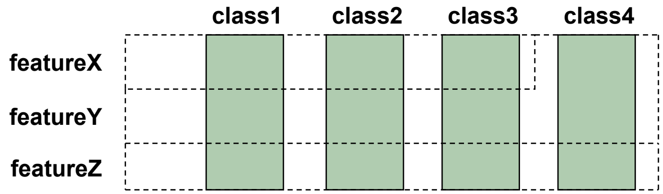
Connecting the Dots...

- A **collaboration** has meaning when it implements a feature
  - ever add a new feature to an existing OO program?
  - several existing classes may be refined
  - several new classes may be added

Synthesis Paradigm

Program P = **featureZ • featureY • featureX**

Note: each feature updates multiple classes



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 are reusable
  - that's the connection with features, product-lines
  - features are reusable – so too must be their implementations

Design is the Key

  - 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

Genesis + Avoca

The First Generation

GenVoca (1988,1992)

- Equates constants, functions with features
- 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

- A **domain model** or **product-line model** or **GenVoca model M**
  - set of constants (base programs)
  - functions (program refinements)

M = { f, h, ... i, j, ... }

Function Composition

- Multi-featured applications are **expressions**

app1 = i • f — application with features f and i

app2 = j • h — application with features h and j

app3 = i • j • f — your turn...

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

---

Expression Optimization

- Constants, functions represent both a feature and its implementation
  - different functions can be different implementations of the *same* feature

$\mathbf{k}_1 \bullet \mathbf{x}$ `// adds k with implementation #1 to x`
$\mathbf{k}_2 \bullet \mathbf{x}$ `// 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…
- See:
  - Batory, et al. "Design Wizards and Visual Programming Environments for GenVoca Generators". *IEEE TSE*, May 2000.

Generalization of Relational Algebra

- Keys to success of Relational Optimizers
  - expression representations of program designs
  - 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, like relational algebra

AHEAD:
The Next Generation

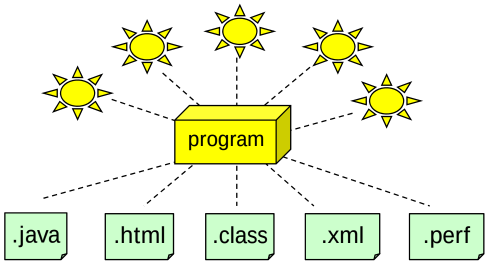
Algebraic **H**ierarchical **E**xpressions for
Application **D**esign

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, ...
- Need 4 insights to capture these ideas

Insight #1: Platonic Forms and Languages

- Each program representation captures different information in different languages



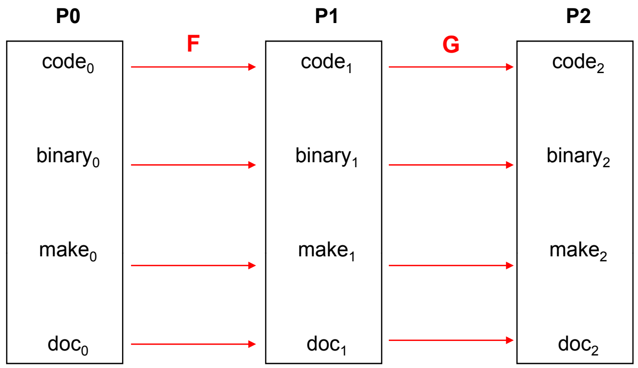
- We want all these representations in a single module

Insight #2: 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

Vectors and Vector Refinements

- A program is a vector of representations
- Features refine vectors component-wise



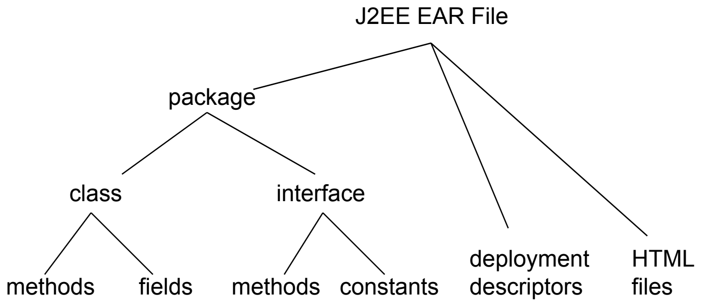
Vector Representations

- **We are reducing program synthesis to vector composition**
- GenVoca model
  - constant P0
  - function F
- **Feature composition = vector composition**
- Still need another idea

$$\begin{array}{ccccc} \mathbf{P1} & & \mathbf{F} & \bullet & \mathbf{P0} \\ \text{code}_1 & & \Delta\text{code}_F & \bullet & \text{code}_0 \\ \text{binary}_1 & = & \Delta\text{binary}_F & \bullet & \text{binary}_0 \\ \text{make}_1 & & \Delta\text{make}_F & \bullet & \text{make}_0 \\ \text{doc}_1 & & \Delta\text{doc}_F & \bullet & \text{doc}_0 \end{array}$$

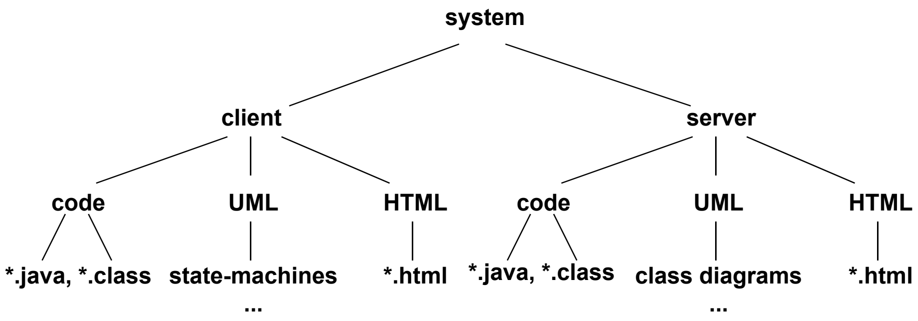
Insight #3: Generalize Modularity

- A **module** is a containment hierarchy of related artifacts



- Generalize module hierarchies to arbitrary depth, contents

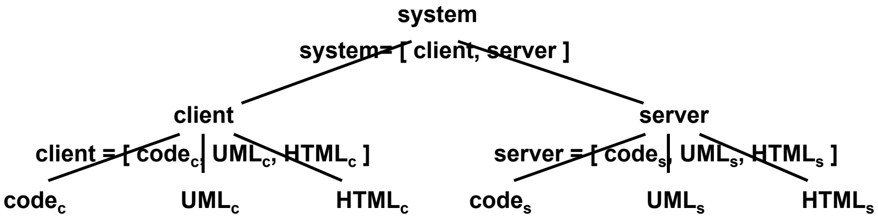
Modularization of Multiple Programs



Modules contain all needed representations of a system

Modules are Nested Vectors

- Program as vector idea recurses: each subrepresentation can itself be a vector



system

system = [client, server]

client

client = [code_c, UML_c, HTML_c]

server

server = [code_s, UML_s, HTML_s]

code_c UML_c HTML_c code_s UML_s HTML_s

- Module is a (nested) vector
- Name of a subrepresentation is unique; it defines its index position in a vector

---

Law of Composition

- Consider base program P and refinement R:

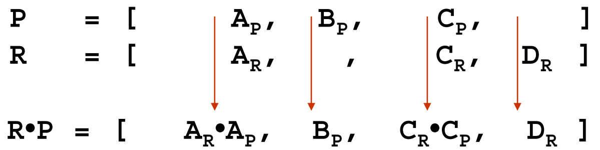
$$P = [\ A_P,\ B_P,\ C_P,\ \ \ ]$$
$$R = [\ A_R,\ \ ,\ C_R,\ D_R\ ]$$

  - implicit vector padding with blanks
  - base programs have nulls (∅)
  - refinements have identity functions (i)

- What is R ● P ?

Law of Composition

- R • P is:

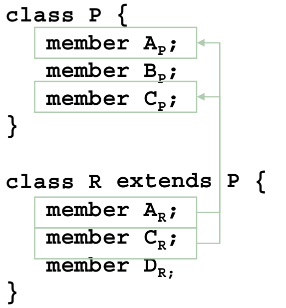


$$P\ \ = [\ A_P,\ B_P,\ C_P,\ \ \ ]$$
$$R\ \ = [\ A_R,\ \ ,\ C_R,\ D_R\ ]$$
$$R\bullet P = [\ A_R\bullet A_P,\ B_P,\ C_R\bullet C_P,\ D_R\ ]$$

- **Says how composition distributes over modularization**
- Do you recognize this law?

Inheritance!

class representation

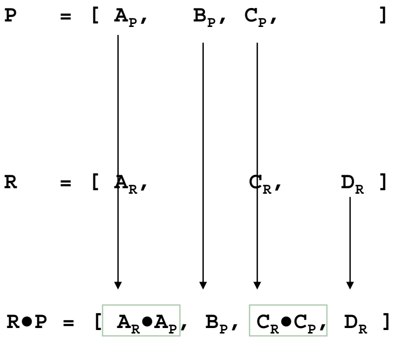


class P {
  member A_P;
  member B_P;
  member C_P;
}

class R extends P {
  member A_R;
  member C_R;
  member D_R;
}

class R●P extends R {}

vector representation

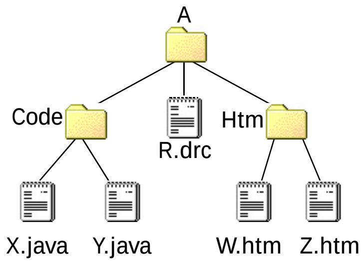


$$P\ \ = [\ A_P,\ \ B_P,\ C_P,\ \ \ ]$$
$$R\ \ = [\ A_R,\ \ \ C_R,\ D_R\ ]$$
$$R\bullet P = [\ \boxed{A_R\bullet A_P},\ B_P,\ \boxed{C_R\bullet C_P},\ D_R\ ]$$

Simple Implementation

- Module hierarchies = nested vectors

directory



vector

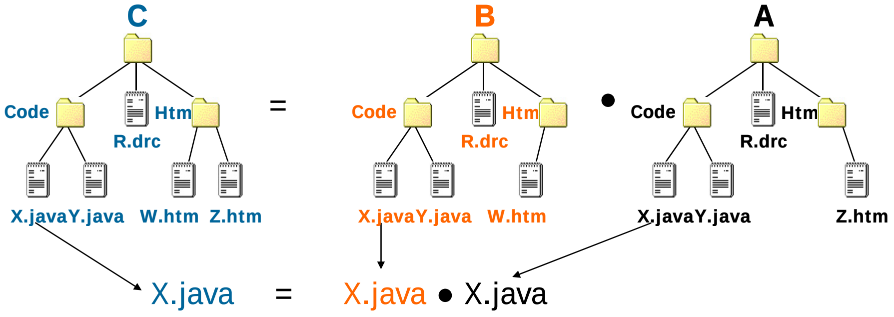
A = [Code, R.drc, Htm]

Code = [X.java, Y.java]

Htm = [W.htm, Z.htm]

Simple Implementation

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



X.java = X.java ● X.java

Simple Theory

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

C = B ● A

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

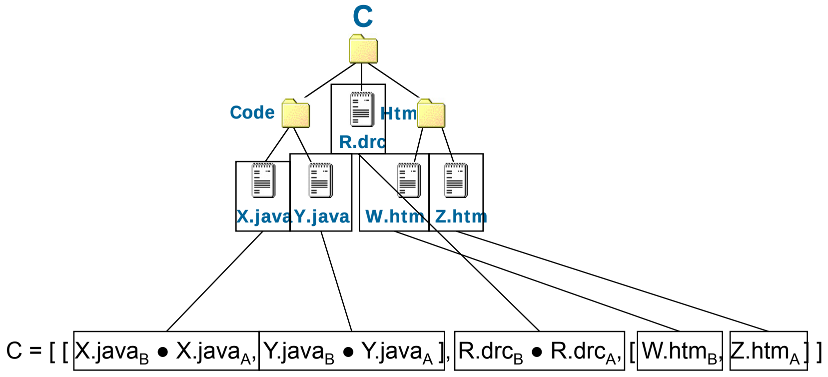
$= [\ \text{Code}_B \bullet \text{Code}_A, \text{R.drc}_B \bullet \text{R.drc}_A, \text{Htm}_B \bullet \text{Htm}_A\ ]$

$= [\ [\ \text{X.java}_B, \text{Y.java}_B\ ] \bullet [\ \text{X.java}_A, \text{Y.java}_A\ ], \text{R.drc}_B \bullet \text{R.drc}_A, [\ \text{W.htm}_B,\ ] \bullet [\ , \text{Z.htm}_A\ ]\ ]$

$= [\ [\ \text{X.java}_B \bullet \text{X.java}_A, \text{Y.java}_B \bullet \text{Y.java}_A\ ], \text{R.drc}_B \bullet \text{R.drc}_A, [\ \text{W.htm}_B, \text{Z.htm}_A\ ]\ ]$

Note!

- Each expression defines an artifact to be produced



$C = [\ [\ \boxed{\text{X.java}_B \bullet \text{X.java}_A}, \boxed{\text{Y.java}_B \bullet \text{Y.java}_A}\ ], \boxed{\text{R.drc}_B \bullet \text{R.drc}_A}, [\ \boxed{\text{W.htm}_B}, \boxed{\text{Z.htm}_A}\ ]\ ]$

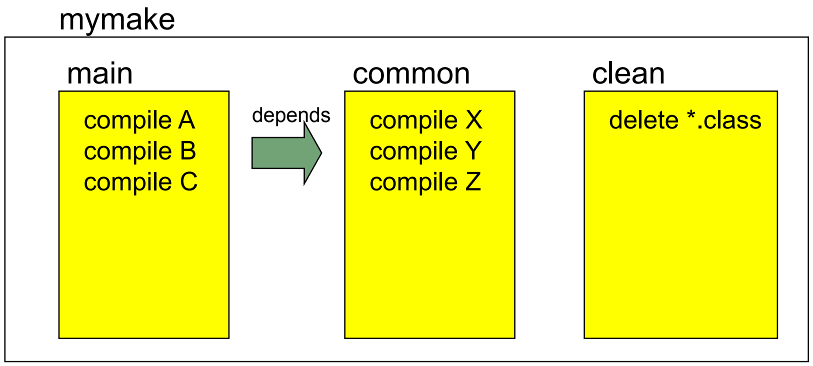
Polymorphism...

- Composition operation ● is **polymorphic**
  - law of composition says how vectors are composed
  - different implementation of ● for each representation
    - » ● for code
    - » another ● 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?????

Makefile



mymake

main

compile A
compile B
compile C

depends

common

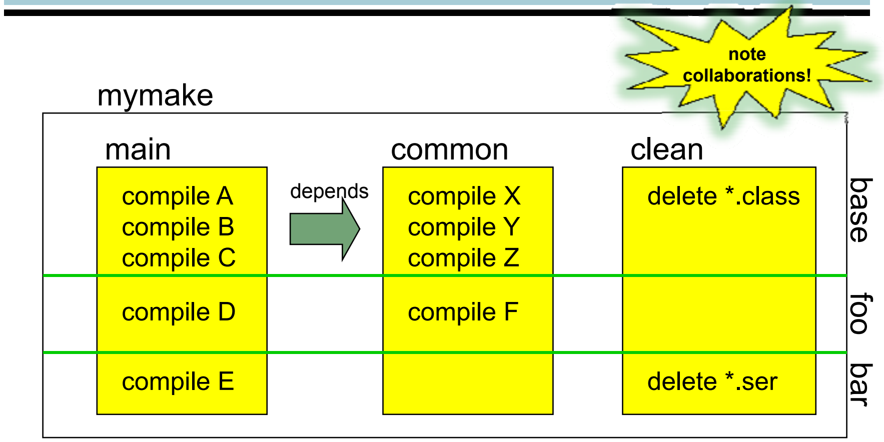
compile X
compile Y
compile Z

clean

delete *.class

command line> make main

Makefile Refinements



note collaborations!

mymake

main

compile A
compile B
compile C

compile D

compile E

depends

common

compile X
compile Y
compile Z

compile F

clean

delete *.class

delete *.ser

base

foo

bar

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

Makefiles

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

```
class myMake {
   void main {
   { ...

   }
   void common {
   ...

   }
   ...
}
```

Insight #4: Principle of Uniformity

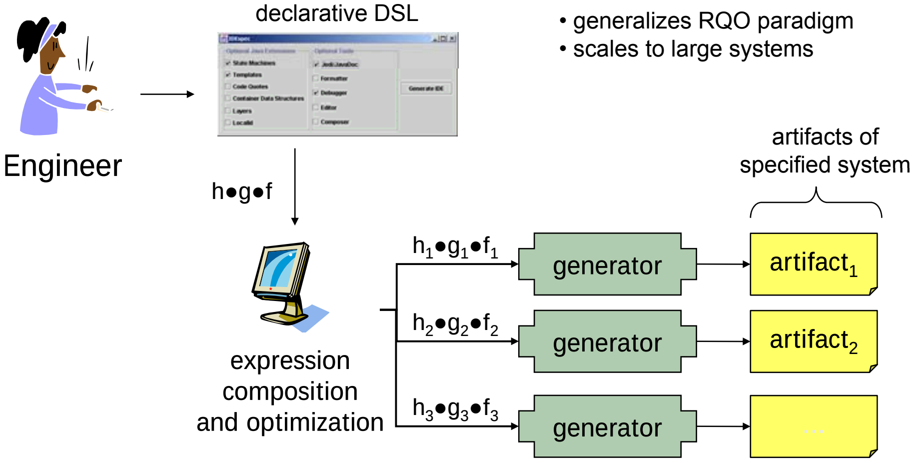
- Treat all artifacts equally, as objects or classes
  - create analog in OO representation
- Refine non-code representations same as code representations
- That is, you can refine any artifact
  - understand it as an object, collection of objects, or classes
- **We are creating a theory of information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

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 step-wise refinement scales without bounds...
- Features 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

Product Member Synthesis Overview

- generalizes RQO paradigm
- scales to large systems



Recommended Readings

- Batory, O'Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. ACM *TOSEM,* October 1992.
- Batory, Sarvela, Rauschmayer. “Scaling Step-Wise Refinement”. *IEEE TSE, June 2004.*
- Batory, Johnson, MacDonald, von Heeder. “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”. *ACM TOSEM*, April 2002.
- Batory, Chen, Robertson, Wang. “Design Wizards and Visual Programming Environments for GenVoca Generators“. *IEEE TSE*, May 2000.
- Batory, Singhal, Thomas, Sirkin. “Scalable Software Libraries”. *ACM SIGSOFT 1993*.
- Batory. “Concepts for a Database System Compiler”. *ACM PODS 1988*.
- Börger, Schulte. “Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation”. MFCS 1998.
- Baxter. “Design Maintenance Systems”. *CACM*, April 1992.
- Czarnecki, Eisenecker. *Generative Programming – Methods, Tools and Applications.* Addison-Wesley 2000.

Recommended Readings

- Czarnecki, Bednasch, Unger, Eisenecker. “Generative Programming for Embedded Software: An Industrial Experience Report”. *GPCE 2002.*
- Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- Ernst. “Higher-Order Hierarchies*”. ECOOP 2003.*
- Garlan, Allen, Ockerbloom. “Architectural Mismatch or Why it is hard to build Systems out of existing parts”. *ICSE 1995*.
- Flatt, Krishnamurthi, Felleisen. “Classes and Mixins”. *ACM POPL* 1998.
- Harrison, Ossher. “Subject-Oriented Programming (A Critique of Pure Objects)”. *OOPSLA 1993*.
- Kang, et al. “Feature Oriented Domain Analysis Feasibility Study”. SEI 1990.
- Kang, et al. “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures”. *Annals of Software Engineering 1998*, 143-168.
- Kiczales, et al. “An Overview of AspectJ”. *ECOOP 2001*.

Recommended Readings

- Lieberherr. *Adaptive Object-Oriented Software.* PWS publishing, 1995.
- Mezini, Lieberherr. “Adaptive Plug-and-Play Components for Evolutionary Software Development”. *OOPSLA 1998*.
- Mezini, Ostermann. “Conquering Aspects with Caesar”. *AOSD 2003*.
- Mezini, Ostermann. “Variability Management with Feature-Oriented Programming and Aspects”. *SIGSOFT 2004*.
- McDirmid, Flatt, and Hsieh. “Jiazzi: new-Age Components for Old-Fashioned Java”. *OOPSLA 2001*.
- Ossher and Tarr. “Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software.” *CACM* October 2001.
- Ossher and Tarr. “Multi-dimensional separation of concerns and the Hyperspace approach”. In *Software Architectures and Component Technology* (M. Aksit, ed.), 2002
- Reenskaug, et al. “OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems”. *Journal of OO Programming*, 5(6): October 1992.

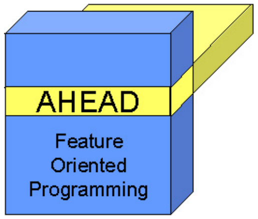
Recommended Readings

- Simonyi. “The Death of Computer Languages, the Birth of Intentional Programming”. *NATO Science Committee Conference*, 1995.
- Smaragdakis, Batory. “Implementing Layered Designs with Mixin Layers”. *ECOOP 1998*.
- Smaragdakis, Batory. “Scoping Constructs for Program Generators”. *GCSE 1999*.
- Smaragdakis, Batory. “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs “. *ACM TOSEM* April 2002.
- Tarr, et al. “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. *ICSE 1999*.
- Van Hilst, Notkin. “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA 1996*.

The AHEAD Tool Suite

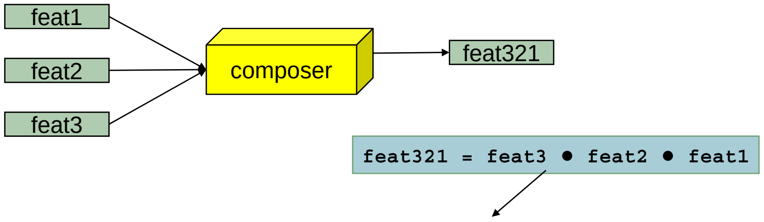
Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy



Composer Tool

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



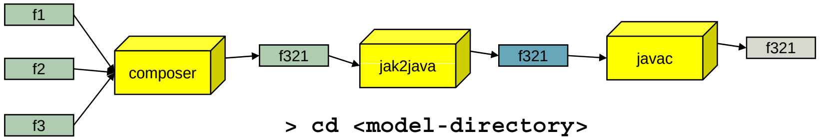
```
> composer -target=feat321 feat1 feat2 feat3
```

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...

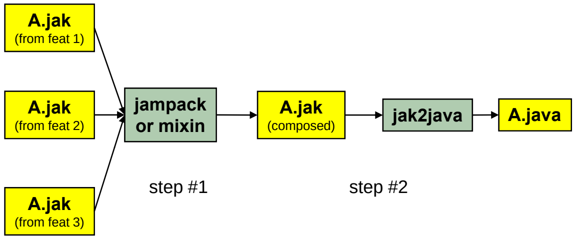
- Besides `composer`
  - `jak2java` – translates Jak files to Java files
  - `javac` – javac compiler
  - `reform` – Jak or Java file formatter/pretty-printer
  - others...



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

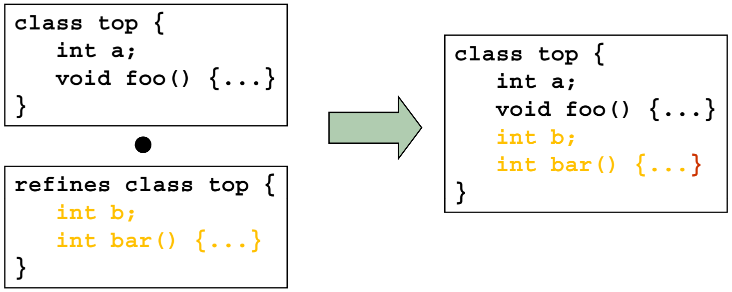
Jak-File Composition Tools

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



`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() {...}
}
```

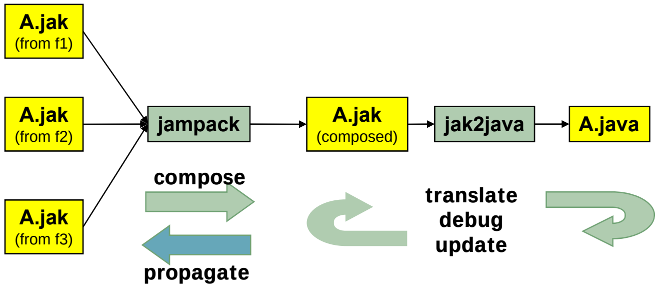
●

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

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

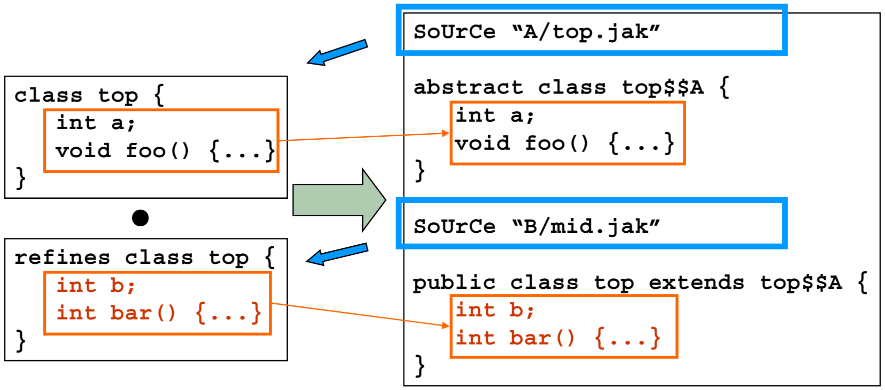
`jampack`

- `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 and its refinements as an inheritance hierarchy



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

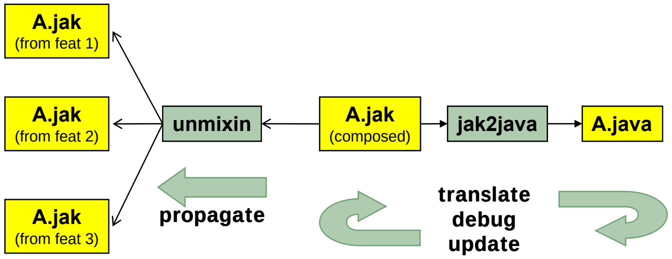
●

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

```
SoUrCe "A/top.jak"

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

SoUrCe "B/mid.jak"

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

`unmixin`

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



A.jak (from feat 1)

A.jak (from feat 2)

A.jak (from feat 3)

unmixin

A.jak (composed)

jak2java

A.java

propagate

translate
debug
update

Composable Representations

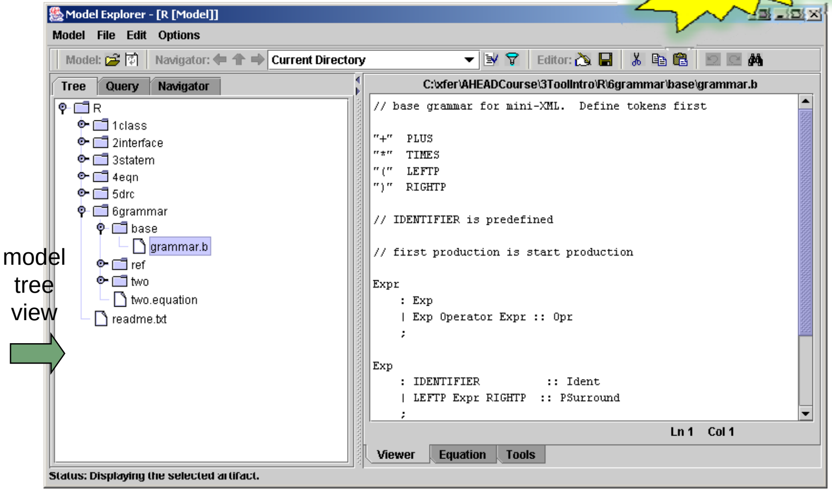
- Current list...
  - *.jak – extended Java files (Jakarta)
    - class
    - interface
    - state machine (ex: embedded DSL)
  - *. equation – named expression files
  - *. b – grammar files
  - *. drc – design rule files
  - others...

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



C:\xfer\AHEADCourse\3ToolIntro\R\6grammar\base\grammar.b

```
// base grammar for mini-XML.  Define tokens first

"+"  PLUS
"*"  TIMES
"("  LEFTP
")"  RIGHTP

// IDENTIFIER is predefined

// first production is start production

Expr
    : Exp
    | Exp Operator Expr :: Opr
    ;

Exp
    : IDENTIFIER          :: Ident
    | LEFTP Expr RIGHTP  :: PSurround
    ;
```

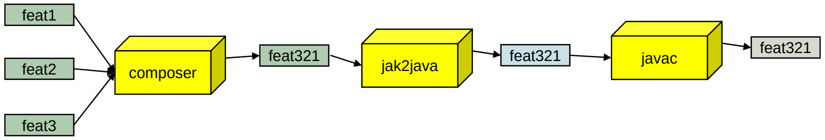
Status: Displaying the selected artifact.

model
tree
view

file view

Cultural Enrichment

- Note algebraic underpinning...

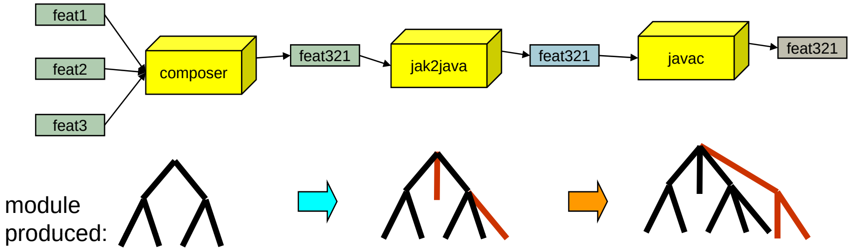


`P =    javac(  jak2java(  f3 • f2 • f1   )  )`

- Same paradigm as AHEAD
  - progressively elaborating a containment hierarchy
  - can optimize expression (not this one...)

Cultural Enrichment

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



- Big picture: lots of operations on AHEAD modules
  - seems that lots of optimizations are possible too...

A Simple Example

to illustrate concepts, tools

Domain of Graph Applications

- A grammar is a simple way to express family of related applications
  - tokens are features
  - sentences are feature compositions

{ undirected | directed } graph { depth-first | breadth-first } search { cycle checking | vertex numbering | connected regions | … }

choose one
choose one
choose at least one

Example Family Members

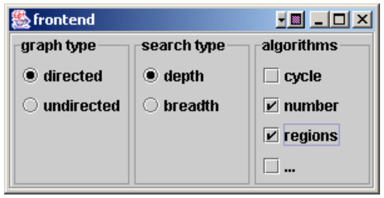
{ **undirected** | directed } graph { depth-first | **breadth-first** } search { cycle checking | **vertex numbering** | connected regions | … }

{ undirected | **directed** } graph { **depth-first** | breadth-first } search { **cycle checking** | vertex numbering | **connected regions** | … }

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


frontend

graph type
● directed
○ undirected

search type
● depth
○ breadth

algorithms
☐ cycle
☑ number
☑ regions
☐ ...

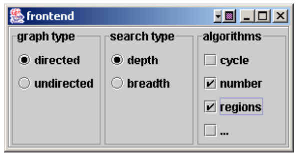
See next lecture on
Verification of Feature Compositions

That's Easy...

- So too is creating the underlying FOP model:

```
Gpl = {
        DIRECTED      – directed graphs       }
        UNDIRECTED    – undirected graphs     }  constants

        BFS           – breadth first search  }
        DFS           – depth first search    }
                                              }  functions
        CYCLE         – cycle checking        }
        NUMBER        – vertex numbering      }
        REGIONS       – connected regions     }
        ...
      }
```

Constructing Applications

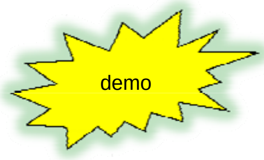

frontend

graph type
● directed
○ undirected

search type
● depth
○ breadth

algorithms
☐ cycle
☑ number
☑ regions
☐ ...

automatic mapping



graph_app = region ● vertex ● dfs ● directed
= vertex ● region ● dfs ● directed

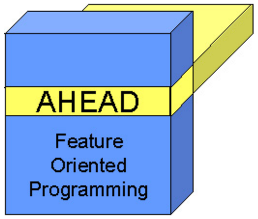
Recommended Readings

- Batory, "A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite*", January 2003.*
- Batory, Sarvela, Rauschmayer, "Scaling Step-Wise Refinement", *IEEE TSE*, June 2004.
- Batory, Cardone, and Smaragdakis, "Object-Oriented Frameworks and Product-Lines". *SPLC* 1999.
- Ernst, "*Higher-Order Hierarchies", ECOOP 2003*.
- Holland, "Specifying Reusable Components Using Contracts", *ECOOP 1992*, 287-308.
- Lee, Siek, and Lumsdaine, "The Generic Graph Component Library", *OOPSLA 1999*.
- Lopez-Herrejon and Batory, "A Standard Problem for Evaluating Product-Line Methodologies", *GCSE 2001.*
- Smaragdakis and Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998*.
- Smaragdakis and Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", *ACM TOSEM*, March 2002.

Verification of Feature Compositions

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

Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy



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!

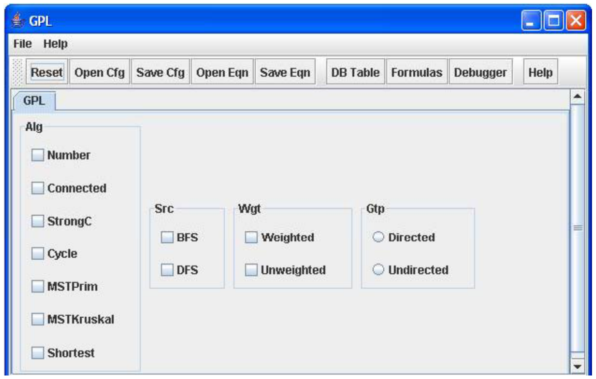
Introduction

- Must verify correctness of 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
  - too easy for users to make mistakes
- 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
  - counterpart to Dell web page
- Constraints propagated as selections are made
  - cannot specify incorrect design
- Can debug model specifications
  - by verifying known properties of feature combinations

Tool Demo



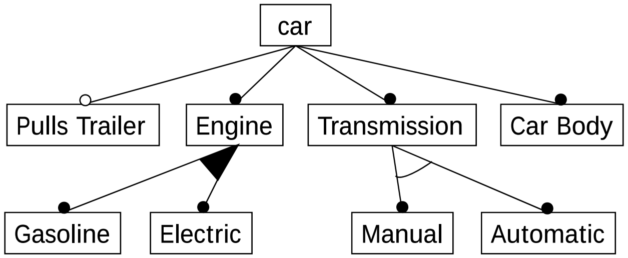
Feature Diagrams and Grammars

(The Theory Behind The Tool)

Grammar ? Feature Diagram

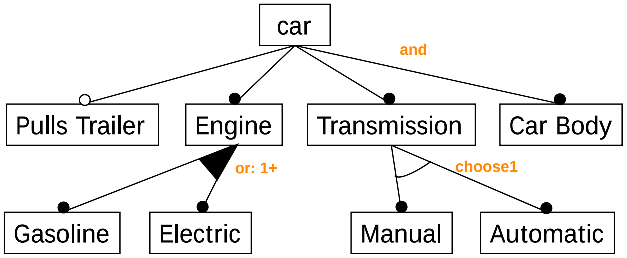
Feature Diagrams

- **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



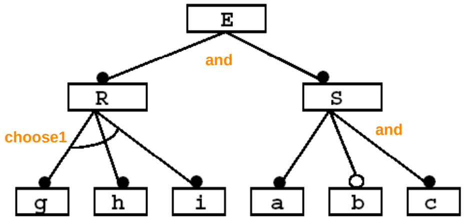
How To Read Feature Diagrams

- Mandatory – features that are required ●
- Optional – features that are optional ○
- And – all subfeatures (children) are selected
- Alternative – only 1 subfeature can be selected
- Or – 1+ or 0+ subfeatures can be selected



Another 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

---

Recall GPL Model

```
Gpl = {
      DIRECTED      – directed graphs       }
      UNDIRECTED    – undirected graphs     }  constants

      BFS           – breadth first search  }
      DFS           – depth first search    }
                                            }  functions
      CYCLE         – cycle checking        }
      NUMBER        – vertex numbering      }
      STRONGC       – strongly connected    }
      ...
     }
```

GPL Grammar

```
Gpl : Alg+ [Src] Wgt Gtp;

Gtp : DIRECTED | UNDIRECTED ;

Wgt : WEIGHTED | UNWEIGHTED ;

Src : DFS | BFS ;

Alg : NUMBER | CONNECTED | STRONGC
    | CYCLE | MSTPRIM | MSTKRUSKAL | SHORTEST ;
```

each token is an AHEAD constant or function

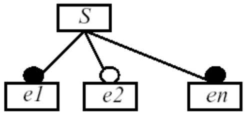
A sentence of this grammar defines a composition of features

```
Prog = NUMBER • CYCLE • BFS • UNWEIGHTED • DIRECTED
```

Mapping of FDs to Grammars

Diagram | **Grammar**

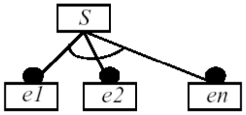
and



S : e1 [e2] en ;

... S ...

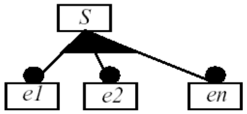
choose1



S : e1 | e2 | en ;

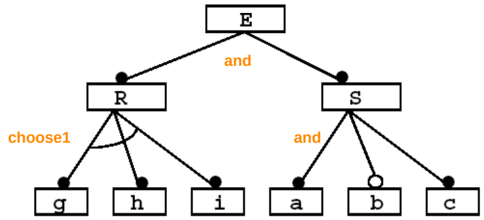
... S+ ...

or: 1+



S : e1 | e2 | en ;

Example: Convert FD to Grammar



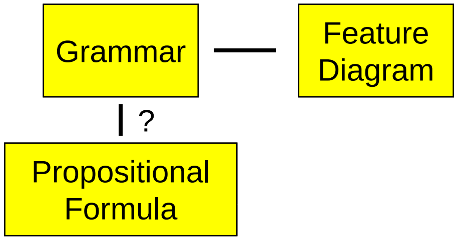
E : R S ;

R : g | h | i ;

S : a [b] c ;

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

Grammars and Propositional Formulas



Propositional Formula

- Set of boolean variables and a propositional logic predicate that constrains values of these variables
- Standard $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ operations
- Nonstandard:
  - atmost1($e_1 \ldots e_k$) – 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 : P1 | ... | Pn ;
- R can be referenced in two ways:

Pattern	Predicate
... R+ ... (choose 1 or more)	$R \Leftrightarrow P1 \vee P2 \vee \ldots \vee Pn$
... R ... (choose 1)	$R \Leftrightarrow (P1 \vee P2 \vee \ldots \vee Pn)$ $\wedge$ atmost1(P1,P2, ..., Pn)

Mapping Patterns to Formulas

- T1 T2 ... Tn :: P

  formula: $P \Leftrightarrow T1 \land P \Leftrightarrow T2 \land \ldots \land P \Leftrightarrow Tn$

- T1 [T2] ... Tn :: Q

  formula: $Q \Leftrightarrow T1 \land T2 \Rightarrow Q \land \ldots \land Q \Leftrightarrow Tn$

Example: Grammars to Formulas

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

```
E : R S ;

R : g | h | i ;

S : a [ b ] c ;
```

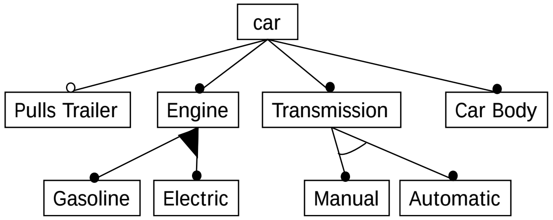
grammar

$$
\begin{aligned}
& E \Leftrightarrow R \land E \Leftrightarrow S \\
\land\ & \\
& R \Leftrightarrow (g \lor h \lor i) \land \\
& \quad \text{atmost1}(g,\ h,\ i) \\
\land\ & \\
& S \Leftrightarrow a \land b \Rightarrow S \land S \Leftrightarrow c \\
\land\ & \\
& E
\end{aligned}
$$

propositional formula

A sentence of E satisfies the propositional formula
and vice versa

Last Example



$$
\begin{aligned}
& \mathbf{Car \Leftrightarrow CB \land Car \Leftrightarrow Tr \land Car \Leftrightarrow Eng \land Pt \Rightarrow Car} \\
\land\ & \\
& \mathbf{Tr \Leftrightarrow (Auto \lor Man) \land atmost1(Auto,Man)} \\
\land\ & \\
& \mathbf{Eng \Leftrightarrow (Ele \lor Gas)} \\
\land\ & \\
& \mathbf{Car}
\end{aligned}
$$

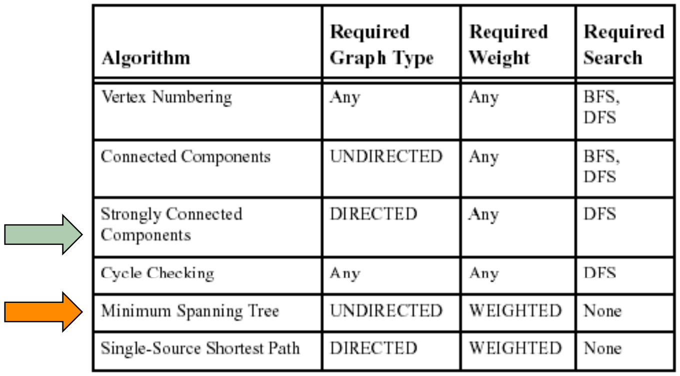
Recap

- 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, add the formula

$$i \lor b \Rightarrow \neg (b \land i)$$

Example: Additional Constraints in GPL

- Straight from Graph Algorithm Text



Algorithm	Required Graph Type	Required Weight	Required Search
Vertex Numbering	Any	Any	BFS, DFS
Connected Components	UNDIRECTED	Any	BFS, DFS
Strongly Connected Components	DIRECTED	Any	DFS
Cycle Checking	Any	Any	DFS
Minimum Spanning Tree	UNDIRECTED	WEIGHTED	None
Single-Source Shortest Path	DIRECTED	WEIGHTED	None

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;
```

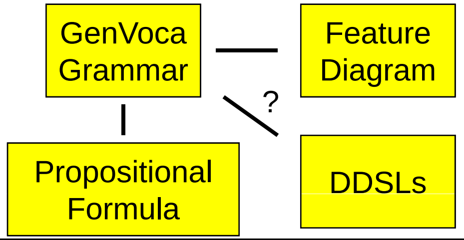
grammar

constraints

Recap

- An AHEAD 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
- See: Forbus and de Kleer, *Building Problem Solvers*, MIT Press 1993.
- BDDs (Binary Decision Diagrams) are also popular

Debugging Feature Models

very useful model debugging aid

Debugging Feature Models

- We know features A and B are compatible
  - let P_{model} be the predicate of our feature model
  - $P_{model} \wedge A \wedge B$ must be satisfiable

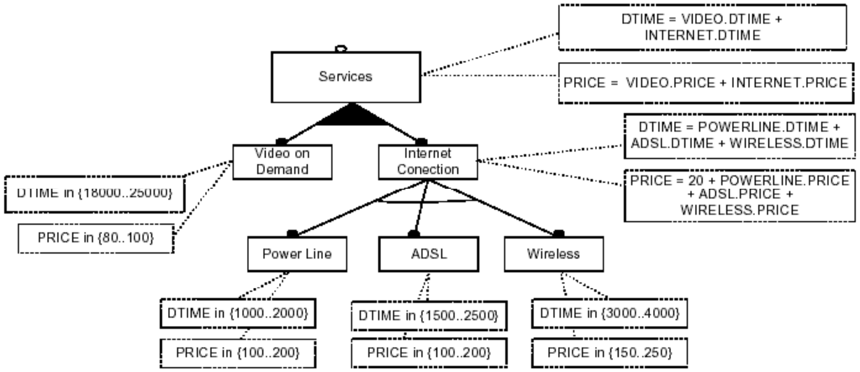
    that is, is there a product that has both A and B?
- **Satisfiability (SAT) Solver**
  - off-the-shelf tool that automatically determines if a boolean predicate is satisfiable
  - very efficient
- Basis for feature model debugging
  - provide a script of compatible, incompatible features and verify that our feature model has these properties
  - solver confirms known properties of a model

Experience

- Has worked well...
- **Use off the shelf constraint solvers**
- Predicates are simple
- 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 Very Exciting!

- Allow features to have additional parameters
  - property lists
- Generalize predicates to include constraints on numeric variables
  - select product that maximizes/minimizes criteria **(performance!)**
  - restrict products based on performance requirements, criteria
  - use standard **Constraint Satisfaction Problem (CSP) Solvers**
  - see: Benavides, et al. "Automated Reasoning on Feature Models", CAISE 2005

Future

- Basic result:

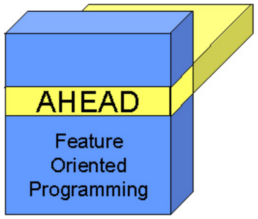
  software design is a satisfiability problem
  - does there exist a system that satisfies the following set of constraints?
- Research: to find optimal system configurations automatically
  - true automatic programming!
  - counterpart to relational query optimizers

Recommended Readings

- Batory and O'Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. *ACM TOSEM*, October 1992.
- Batory and Geraci. “Composition Validation and Subjectivity in GenVoca Generators”, *IEEE TSE*, Feb 1997.
- Batory, “Feature Models, Grammars, and Propositional Formulas”, SPLC 2005.
- Benavides, Trinidad, and Ruiz-Cortes, “Automated Reasoning on Feature Models”, *Conference on Advanced Information Systems Engineering (CAISE)*, July 2005.
- Beuche, Papajewski, and Schroeoder-Preikschat, “Variability Management with Feature Models”, *Science of Computer Programming*, Volume 53, Issue 3, Pages 333-352, December 2004.
- Czarnecki and Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Add.-Wes., 2000.
- Forbus and de Kleer, *Building Problem Solvers*, MIT Press 1993.
- de Jong and Visser, “Grammars as Feature Diagrams”, 2002. `http://www.cwi.nl/events/2002/GP2002/papers/dejonge.pdf`
- Neema, Sztipanovits, and Karsai, “Constraint-Based Design Space Exploration and Model Synthesis”, *EMSOFT 2003*, LNCS 2855, p. 290-305.
- Perry, “The Logic of Propagation in the Inscape Environment”, ACM SIGSOFT 1989.
- Zhang, Gao, Jacobsen, “Towards Just-in-time Middleware Architectures”, AOSD 2005.

Program Refactoring, Program Synthesis, and Model Driven Design

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



Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy

This Lecture

- Sketch where I see
  - automated software design & maintenance is headed
- Essential complexity of software structure
  - is exposed when program construction and design is viewed as a computation
- **Architectural Metaprogramming**
  - programs are values
  - transformations map programs to programs
  - operators map transformations to transformations

Architectural Metaprogramming

- Lies at core of many important areas in software design and maintenance:
  - refactorings are **behavior-preserving** transformations
  - feature-based and aspect-based software synthesis use **behavior-extending** transformations
  - model driven design uses both to map **platform independent models (PIMs)** to **platform specific models (PSMs)**
- Lecture reveals a bigger world in which FOP lies

Relationship of Design to Set Arithmetic

- Is basic to engineering
- **Computer Aided Design (CAD)** tools enable engineers to express designs by adding, subtracting, and transforming volumes from which properties of designs are derived
- Architectural metaprogramming offers a program analog: programs can be added, subtracted, and transformed
  - set arithmetic captures essential design concepts
  - **accidental complexities and limitations of languages, tools, and implementations are abstracted away**

Upcoming Topics – Four “Mini” Talks

- Basics of **Architectural Metaprogramming**
  then reflect on 2006 advances in

- **Program Refactoring**
  - Danny Dig & Ralph Johnson (Illinois)

- **Program Synthesis**
  - Roberto Lopez-Herrejon (Oxford) & Christian Lengauer (Passau)

- **Model Driven Design**
  - Salva Trujillo & Oscar Diaz (Basque Country)

- All topics describe systems that have been built
  - step back and give a simple explanation of their results

#1 ral

Architectural Metaprogramming

- Programs are **values**

- Here is a value
  (Java definition of class C):

```
class C {
   int x;
   void inc() {...}
   ...
}
```

- Here is another value:

```
class D {
   void compute()
   {..}
   ..
}
```

1st Operation: + (Sum)

- Let D =

```
class D {
   void compute()
   {..}
}
```

and C =

```
class C {
   int x;
   void inc() {...}
}
```

- D + C =

```
class D {
   void compute()
   {..}
}
```

```
class C {
   int x;
   void inc() {...}
}
```

Another Example

- Let C1 =

```
class C {
   void comp() {..}
}
```

and C2 =

```
class C {
   int x;
   void inc() {...}
}
```

- C1 + C2 =

```
class C {
   void comp () {..}

}
```

```
class C {

   int x;
   void inc() {...}
}
```

+ (Sum) is Disjoint Set Union

- Has expected properties
  - 0 is identity (null program)

    P = 0 + P = P + 0

  - commutative (because disjoint set union is commutative)

    A + P = P + A

  - associative (because disjoint set union is associative)

    (A + B) + C = A + (B + C)

2nd Operation: – (Sub)

- Subtraction is set difference

  (D + C) – C = D

- Has expected properties:
  - Left associative P – C – D = ((P – C) – D)
  - Not commutative P – C ≠ C – P
  - Identity P – 0 = P

3rd Operation: Distributive Transformations

- **Transformation** is a function that maps programs to other programs
- **Rename(p,q,r)** – in program "p" replace name "q" with "r"

Rename(

```
class C {
   int x;
   void inc() {..x..}
   …
}
```

, C.x, C.z) =

```
class C {
   int z;
   void inc() {.. z ..}
   …
}
```

Another Example

Rename(

```
class D {
   void compute()
   {..}
   ..
}
```

, C.x, C.z) =

```
class D {
   void compute()
   {..}
   ..
}
```

- Called a **fixed point**:
  - a value x such that f(x) = x
- Distributive transformations have lots of fixed points

A Key Property of Distributive Transformations

- Transformations we consider **distribute** over + and –

$$f(A + B) = f(A) + f(B)$$

$$f(C – D) = f(C) – f(D)$$

- Here's an example...

Example of Distributivity

Rename(

```
class D {
   void compute()
   {..}
   ..
}

class C {
   int x;
   void inc() {..x..}
   ...
}
```

, C.x, C.z) =

Rename(

```
class D {
   void compute()
   {..}
   ..
}
```

, C.x, C.z)

\+

Rename(

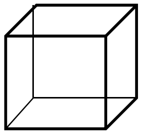
```
class C {
   int x;
   void inc() {..x..}
   ...
}
```

, C.x, C.z)

Structures & Properties

- **Structure** – what are the parts and how are they connected?

cube



a solid bounded by six equal squares, the angle between any two adjacent faces is a right angle.

- **Properties** of structure – attributes derivable from structure
  - surface area = $6E^2$; where E is edge length
  - volume = E^3

Software Analogs

- **Structure** of a program is a **meta expression**

- F ructure

In this lecture,
I focus on program structure.

Results on properties
are presented elsewhere.

ograms –

es

#2: Advances in Program Refactoring

Refactoring

- Is a program transformation that changes the structure of a program, but not its behavior
  - rename methods
  - move method from subclass to superclass
  - ...
- Most design patterns are end-products of refactorings
- Common IDEs (Eclipse, Visual Studio, IntelliJ) have refactoring tools or plug-ins
- Here's an interesting refactoring problem

Evolution of APIs

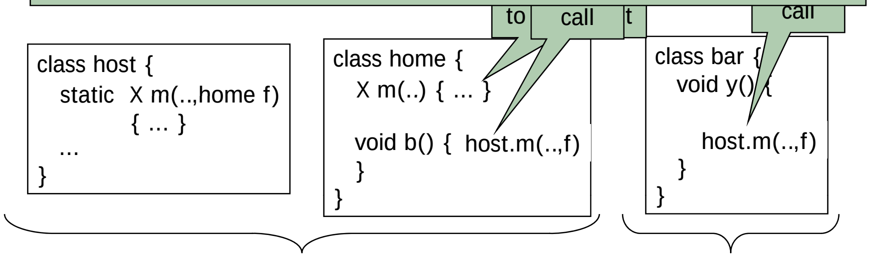
- Use of components (e.g. frameworks, libraries) are common in software development
  - build systems faster and cheaper
- **Application Program Interface (API)** of a component – set of (Java) interfaces and classes that are exported to application developers
  - ideally, APIs don't change, but of course they do!
  - **when APIs change, client code must also change**
  - **very disruptive event in program development**
- Need an easy and safe way to update applications when component's API changes

A Common API Change

- Move Method

Note: although component code changes,
client code must also change

But a component developer doesn't have the client code



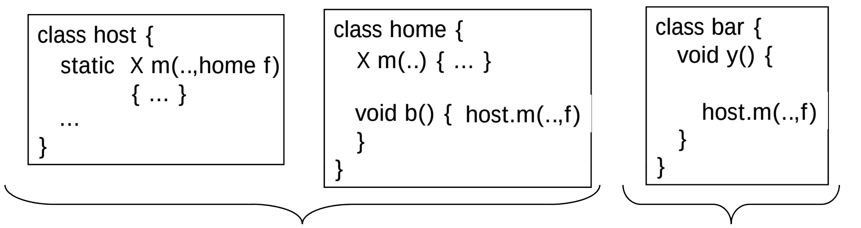
```
class host {
   static  X m(..,home f)
           { … }
   …
}
```

```
class home {
   X m(..) { … }

   void b() {  host.m(..,f)
   }
}
```

```
class bar {
   void y() {

      host.m(..,f)
   }
}
```

This Change is a Meta-Expression

$$P_{new} = \rho \bullet \mu \left(P_{old} \right)$$



```
class host {
   static  X m(..,home f)
           { … }
   …
}
```

```
class home {
   X m(..) { … }

   void b() {  host.m(..,f)
   }
}
```

```
class bar {
   void y() {

      host.m(..,f)
   }
}
```

Other Common API Changes

- Move Field
- Delete Method
  - usually done after a method is renamed or moved
- Change Argument Type
  - ex: replace argument type with its supertype
- Replace Method Call
  - with another that is semantically equivalent and in the same class
- Lots of others...
  - preliminary work suggests all can be written as meta expressions

---

Result

- Dig & Johnson paper:

  "How do APIs Evolve: A Story of Refactoring"
  Jour. Software Maintenance & Evolution:
  Research & Practice 2006

- Manually analyzed change logs, documentation, etc. of different versions of 5 medium to large systems (50K to 2M LOC)
  - Eclipse, Struts, JHotDraw...
- Found over 80% of API changes are refactorings
  - means LOTS of tedious & error-prone updates can be **automated**
  - explain elegance of their solution using architectural metaprogramming

In the Future

- Programmers will use advanced IDEs that “mark” API classes, methods, fields
  - only way marked elements can change is by refactorings (β)
  - “private” component edits modeled by transformations (**e**)

$$\beta3\bullet e3\bullet e2\bullet\beta2\bullet\beta1\bullet e1\left(\boxed{\text{version 0}}\right) = \boxed{\text{version 1}}$$

$\beta =$

transformations to be applied to update client code w.r.t. changes in API

- API updates β is a projection of changes where “private” edits are removed

---

Client Update Meta-Function U

$$U\left(\boxed{\text{client program}}\right) = \beta\left(\boxed{\text{client program}} - \boxed{\text{version 0}}\right) + \boxed{\text{version 1}}$$

$$U\left(\boxed{\text{client program}}\right) = \beta\left(\boxed{\text{client code}} + \boxed{\text{version 0}} - \boxed{\text{version 0}}\right) + \boxed{\text{version 1}}$$

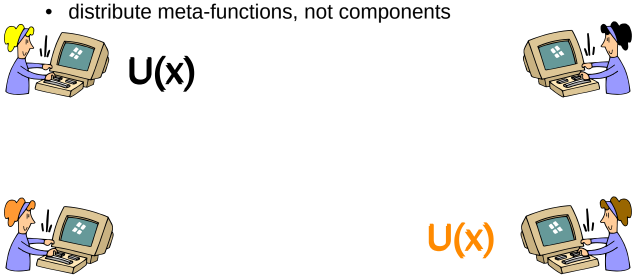
$$= \beta\left(\boxed{\text{client code}}\right) + \boxed{\text{version 1}}$$

this is **not** how result is presented by Dig and Johnson; it is an architectural metaprogramming expression of their results

---

In the Future

- IDEs will be **component evolution calculators**
- IDEs will create update functions like U for distribution
  - distribute meta-functions, not components



- IDEs will apply functions to code bases to automatically update them
- Architectural metaprogramming is at the core of this technology

---

#3: Advances in Program Synthesis

Background

- Previous lectures have presented basic ideas on feature modularity and product lines
- But now, let's look inside the structure of features and see how it is related to
  aspect-oriented programming (AOP)
  - find similarities and differences between aspects and features

---

What Are FOP Features?

- If we peer inside features we see familiar ideas popularized by AOP
  - here I use ideas of AOP
- **Introduction** – adds new members to existing classes
  - corresponds to metaprogramming addition
- **Advice** – modifies methods at particular points, called **join points**
  - quantification means advise all parts of a program – distributivity!
  - advice is a distributive transformation
  - advice is behavior-extending not behavior-preserving
- No "subtraction" in AOP or in FOP

Introduction

- Incrementally add new members, classes

Program P

```
class C {
   void foo(){..}
   int i;
   String b;
}

class D {
  String bar;
  int cnt(){..}
}
```

Meta-Algebra Interpretation

`P = C.b + C.foo + C.i + D.bar + D.cnt`

Program P

```
class C {
   void foo(){..}
   int i;
   String b;
}

class D {
  String bar;
  int cnt(){..}
}
```

Advice

- Defined in terms of events called **join points**
  - when method is called
  - when method is executed
  - when a field is updated
  - ...
- **Advice**: when particular join points occur, execute a given piece of code
- Although advice has a “dynamic” interpretation, we can give it a “static” metaprogramming interpretation

---

Advice Example

Program P

```
class C {
   int i,j;
   void setI (int x){ i=x;                }
   void setJ (int x){ j=x;                }
}

after(): execution (void C.set*(..))
   { print(“hi”); }
```

Meta-Algebra Interpretation

Program P

```
class C {
   int i,j;
   void setI'(int x){ i=x;             }
   void setJ'(int x){ j=x;             }
}

after(): execution (void C.set*(..))
   { print(“hi”); }
```

```
P =      C.i  +    C.j   +    C.setI’ +    C.setJ’
```

Structure of Features

- Features are metaprogramming functions that:
  - advise (**a**) an existing program (**x**)
  - introduce new terms (**i**)

$$F(x) = \overset{\text{adds new code}}{i_f} + \overset{\text{changes existing code to integrate new functionality}}{a_f}\ (x)$$

- Composition:

$$G(\,F(\,B\,)\,) = i_g + a_g\,(\,i_f + a_f\,(\,b\,)\,)$$

In the Future

- Many (narrow) domains will be well-understood
    - know problems, solutions
- Complexity controlled by standardization
    - programs specified declaratively using “standard” features (like Dell)
- Compilers will be **program calculators**
    - inhale source code
    - generate meta-expression, maybe optimize expression
    - evaluate to synthesize program
- Architectural metaprogramming is at core of these technologies

---

A n

Bet gs?

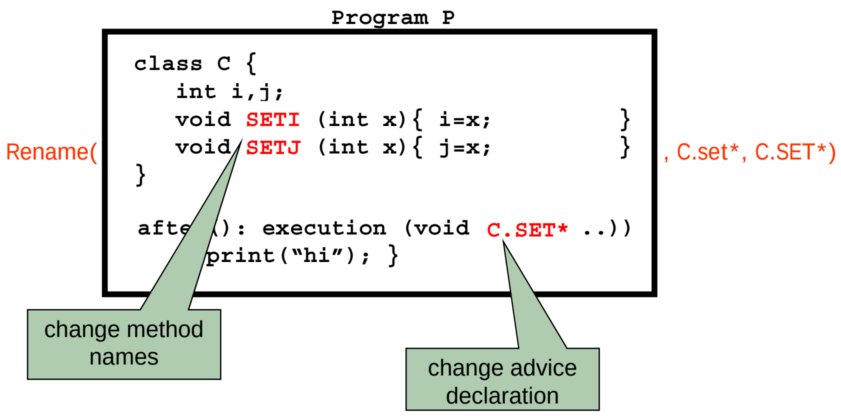
Big Picture

- Refactorings and advice are both transformations
- Suppose I have a refactoring and advice to apply to a program. What does it mean to compose them?
- Advice does not modify a refactoring

  a refactoring is not a language construct;
  there are no join points in a refactoring

- Refactoring can modify programs that include advice

Example



Program P

Rename(

```
class C {
   int i,j;
   void SETI (int x){ i=x;          }
   void SETJ (int x){ j=x;          }
}

afte ): execution (void C.SET* ..))
      print("hi"); }
```

, C.set*, C.SET*)

change method names

change advice declaration

Meta-Algebra

- Remember differential operators in calculus?
  - they transform expressions

$$\frac{\partial(a+b+c)}{\partial y} = \frac{\partial a}{\partial y} + \frac{\partial b}{\partial y} + \frac{\partial c}{\partial y}$$

each term is transformed

- Rename refactoring is similar
  – it transforms each term of a meta expression

$$\beta(\ i + a(x)\ ) = \beta(i) + \beta(a)(\ \beta(x)\ )$$

Homomorphisms

- Such a mapping is an example of a:

homomorphism

  - structure-preserving map between algebras

- Grounded in **Category Theory**
  - theory of mathematical structures and their relationships
  - more later...

How Meta-Calculation Proceeds

Rename(

```
Program P
class C {
   int i,j;
   void SETI (int x){ i=x;          }
   void SETJ (int x){ j=x;          }
}

after(): execution (void C.SET*(..))
   { print("hi"); }
```

, C.set*, C.SET*)

`HI ( C.i + C.j + C.SETI + C.SETJ )`

Recap

- Architectural meta-algebra is getting more interesting
  - refactorings are operators on meta expressions that have higher-precedence than advice

- The rewrite rules for a refactoring **R** is:

$$R(\ a + b\ ) = R(\ a\ ) + R(\ b\ )$$

$$R(\ a - b\ ) = R(\ a\ ) - R(\ b\ )$$

$$R(\ a \bullet b\ ) = R(\ a\ ) \bullet R(\ b\ )$$

Another Interesting Question

What does AspectJ really do?

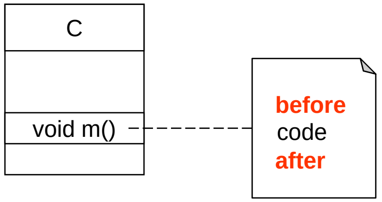
Skip

Basic Differences of FOP and AspectJ

- Aspects don't compose
  - to this day, you cannot express all aspect files as a composition of simpler aspect files
  - reason: rules for ordering around, before, after advice are incomprehensible
  - see AspectJ documentation
- Unbounded quantification
  - AspectJ applies advice after **all** introductions have been made
  - FOP applies advise at different stages of program development
- Why does AspectJ use unbounded quantification?

Tutorial – Method Refinement

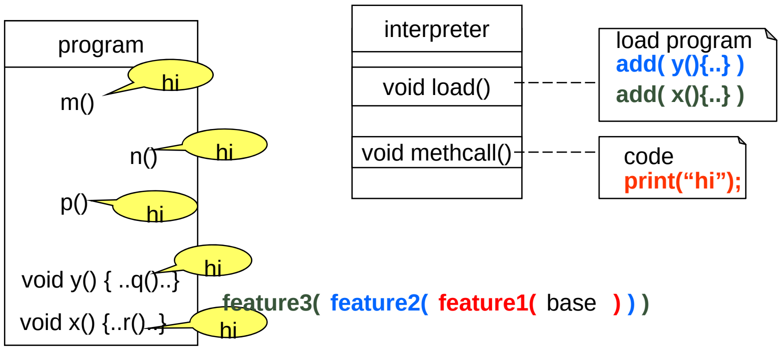
- Features refine individual methods by before, around, after advice



feature(base **)**

Aspects Originate From MetaClasses ~1990

- Don't think of programs, think of interpreters and **refining** interpreters with new features



feature3(feature2(feature1(base **)))**

Insight

- When you define advise or introductions in AspectJ, you are refining (adding features to) the Java interpreter!
  - effects of advice are PROGRAM WIDE
  - advises entire program (**no matter when introductions are made**)
  - “unbounded” advice basic to AOP
- When you refine a program in FOP
  - effects of advise limited to the current state of a program’s design
  - “bounded advice”
- **Refining programs ≠ refining language interpreters!**
- Historically, incremental software design (e.g., agile programming) never “refines” interpreters, only “programs”

Example of UnBounded Quantification

Program P’

```
class C {
   int i,j,k ;
   void setI’(int x){ i=x;                }
   void setJ’(int x){ j=x;                }
   void setK’(int x){ k=x;                }
}
after(): execution (void set*(..))
   { print(“hi”); }
```

`P’ = hi●(C.k + C.setK + C.i+C.j+C.setI+C.setJ )`

Example of Bounded Quantification

Program P

```
class C {
   int i,j,k ;
   void setI’(int x){ i=x;             }
   void setJ’(int x){ j=x;             }
   void setK (int x){ k=x }
}
after(): execution (void set*(..))
   { print(“hi”); }
```

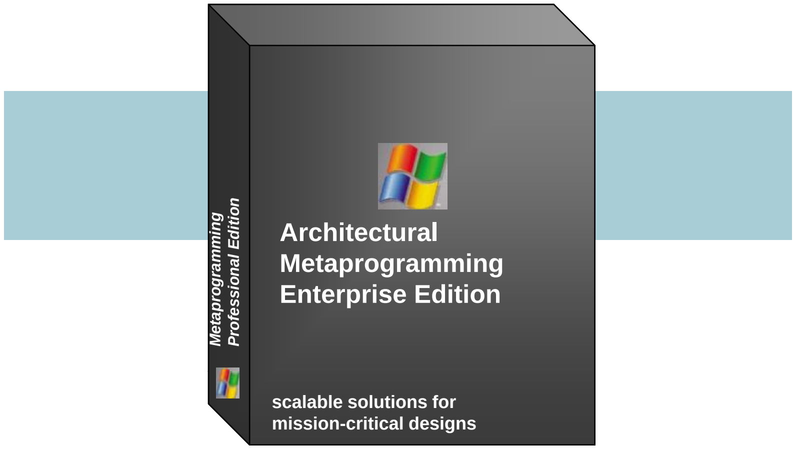
`P = C.k + C.setK + hi●(C.i+C.j+C.setI+C.setJ )`

Different Kinds of Quantification

- May need both because they are doing semantically different things for different purposes
  - bounded advice standard for program synthesis
  - unbounded advice used for **invariants** – program-wide constraints
- Architectural metaprogramming shows these distinctions

Looking Forward

- Notice:
  - refactorings
  - advice
  - introductions
  - modify **structure** of code
    but could also modify **structure** of grammars, makefiles, xml documents, MDD models ... as well
- Generalizing meta-algebra beyond code structures to non-code structures...
  - theory applies to all documents that can be synthesized



Introduction

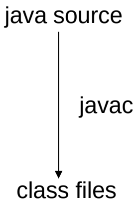
- **Model Driven Design (MDD)** is an emerging paradigm for software creation
  - uses **domain-specific languages (DSL)**
  - encourages automation
  - exploits data exchange standards
- Model is written in a DSL
  - captures particular details of program's design
  - several models are needed to specify a program
  - **models can be derived from other models by transformations**
  - program synthesis is transforming high-level models into executables (which are also models)
  - Bezivin "Everything is a Model"

MDD Tools

- OMG's **Model Driven Architecture (MDA)**
  - define models in terms of UML
  - transform models using graph transformations (QVT)
- First and best works I've seen is Vanderbilt's **Model Integrated Computing (MIC)** and Tata's MDD work and **MasterCraft** tools
- Lots of other groups:
  - Eclipse
  - Microsoft's Software Factories
  - Borland
  - ...

Metaprogramming Connection

- MDD embraces concept that program development is a **computation**
  - **claim**: MDD is a metaprogramming paradigm
  - models are values
  - transformations are functions that map models to models
- Common example



javac transforms java source to class files

Interesting Question

- If javac is a transformation, is it distributive?

javac is not distributive!

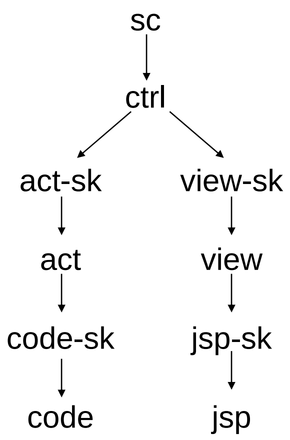
Although there is research by Ancona et. al. on

Separate Class Compilation

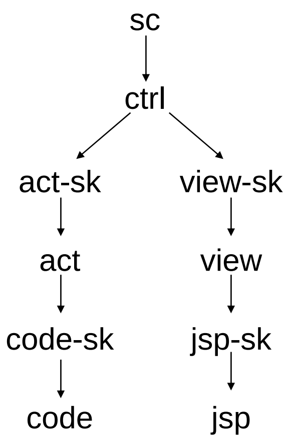
that makes it so…

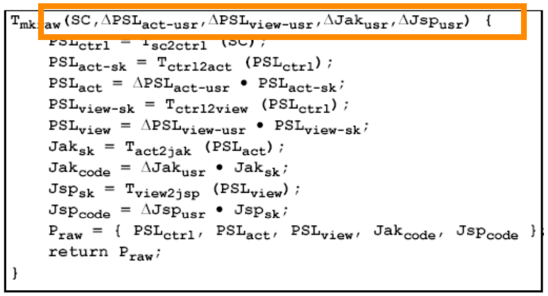
More Typical MDD Example: PinkCreek

- Work with S. Trujillo and O. Diaz
- Portlet is a web component
- PinkCreek is an MDD case study for synthesizing portlets
- Uses transformations to map an annotated state chart (SC) to different representations (Java, JSP code)



Portlet Synthesis Metaprogram





```
$T_{mkraw}$($SC$,$\Delta PSL_{act-usr}$,$\Delta PSL_{view-usr}$,$\Delta Jak_{usr}$,$\Delta Jsp_{usr}$) {
    $PSL_{ctrl}$ = $T_{sc2ctrl}$ ($SC$);
    $PSL_{act-sk}$ = $T_{ctrl2act}$ ($PSL_{ctrl}$);
    $PSL_{act}$ = $\Delta PSL_{act-usr}$ • $PSL_{act-sk}$;
    $PSL_{view-sk}$ = $T_{ctrl2view}$ ($PSL_{ctrl}$);
    $PSL_{view}$ = $\Delta PSL_{view-usr}$ • $PSL_{view-sk}$;
    $Jak_{sk}$ = $T_{act2jak}$ ($PSL_{act}$);
    $Jak_{code}$ = $\Delta Jak_{usr}$ • $Jak_{sk}$;
    $Jsp_{sk}$ = $T_{view2jsp}$ ($PSL_{view}$);
    $Jsp_{code}$ = $\Delta Jsp_{usr}$ • $Jsp_{sk}$;
    $P_{raw}$ = { $PSL_{ctrl}$, $PSL_{act}$, $PSL_{view}$, $Jak_{code}$, $Jsp_{code}$ };
    return $P_{raw}$;
}
```

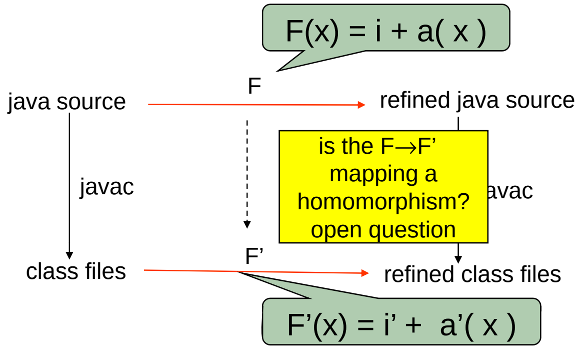
Example of using transformations to derive different models or representations of a program

Another Interesting Question...

As FOP and MDD are both
metaprogramming paradigms,
how do they combine?

In the Future

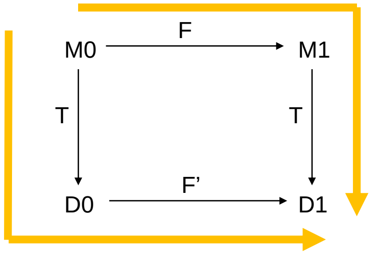
- Features “extend” or “refine” models
- An example:



Fundamental Relationship

- Relationship between transformations that derive models and those that refine models

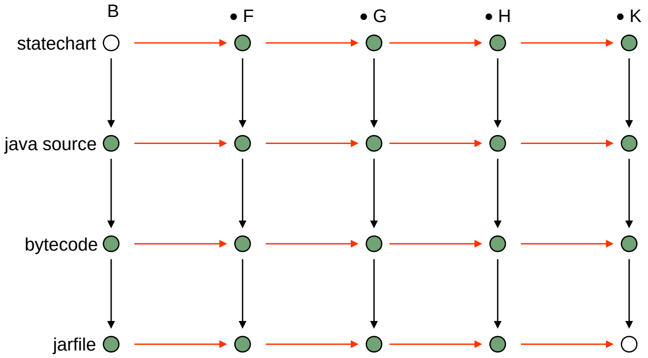
Pushout



$$\begin{tikzcd}
M0 \arrow[r, "F"] \arrow[d, "T"'] & M1 \arrow[d, "T"] \\
D0 \arrow[r, "F'"] & D1
\end{tikzcd}$$

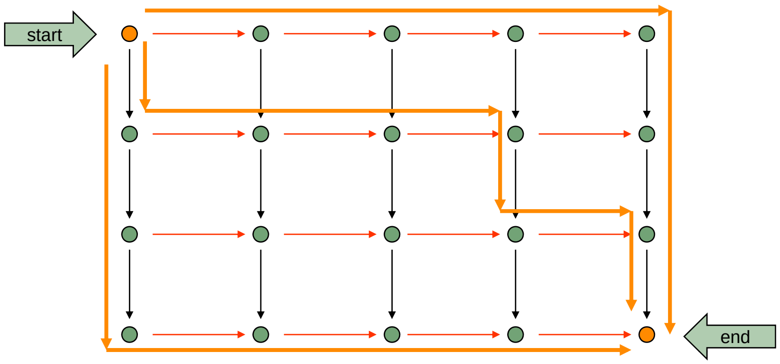
Commuting Diagram

How Commuting Diagrams are Created



- Begin with derivation of representations of base program
- Each feature refines each of these representations

Property of Commuting Diagrams

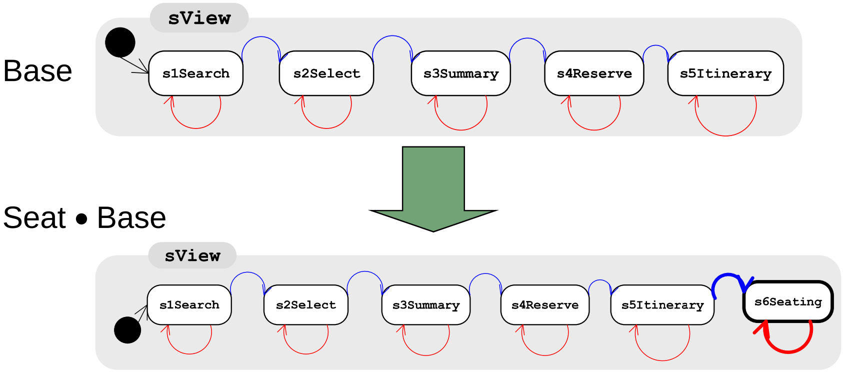


- Given model in upper left, often want to compute model in lower right
- Any path from upper left to lower right should produce same result
- Each path represents a different metaprogram that produces same result

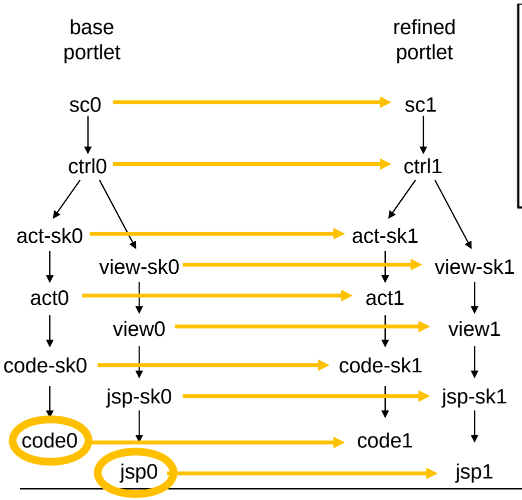
---

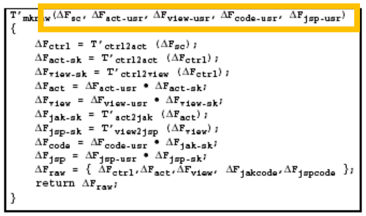
Example: Refining State Charts in PinkCreek

- Features refine state charts by adding new states, transitions, annotations, etc.



How State Charts are Refined in PinkCreek

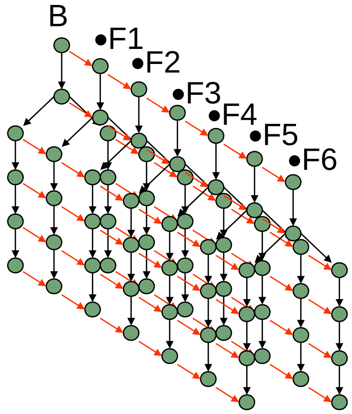




```
$T'_{mkraw}(\Delta F_{sc}, \Delta F_{act-usr}, \Delta F_{view-usr}, \Delta F_{code-usr}, \Delta F_{jsp-usr})$
{
    $\Delta F_{ctrl} = T'_{ctrl2act}(\Delta F_{sc})$;
    $\Delta F_{act-sk} = T'_{ctrl2act}(\Delta F_{ctrl})$;
    $\Delta F_{view-sk} = T'_{ctrl2view}(\Delta F_{ctrl})$;
    $\Delta F_{act} = \Delta F_{act-usr} \bullet \Delta F_{act-sk}$;
    $\Delta F_{view} = \Delta F_{view-usr} \bullet \Delta F_{view-sk}$;
    $\Delta F_{jak-sk} = T'_{act2jak}(\Delta F_{act})$;
    $\Delta F_{jsp-sk} = T'_{view2jsp}(\Delta F_{view})$;
    $\Delta F_{code} = \Delta F_{code-usr} \bullet \Delta F_{jak-sk}$;
    $\Delta F_{jsp} = \Delta F_{jsp-usr} \bullet \Delta F_{jsp-sk}$;
    $\Delta F_{raw} = \{ \Delta F_{ctrl}, \Delta F_{act}, \Delta F_{view}, \Delta F_{jakcode}, \Delta F_{jspcode} \}$;
    return $\Delta F_{raw}$;
}
```

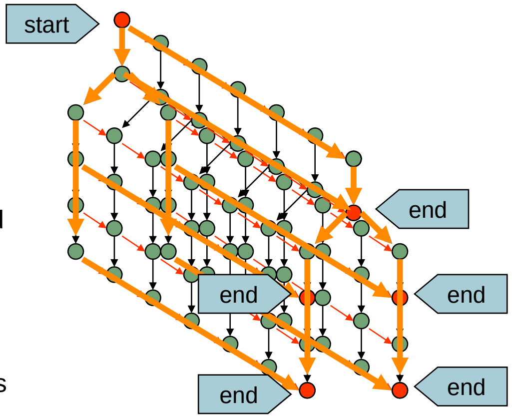
Commuting Diagrams in PinkCreek

- Features map space of artifacts by refining them
- Composing features sweeps out the commuting diagrams to traverse to synthesize portlet representations



Portlet Synthesis

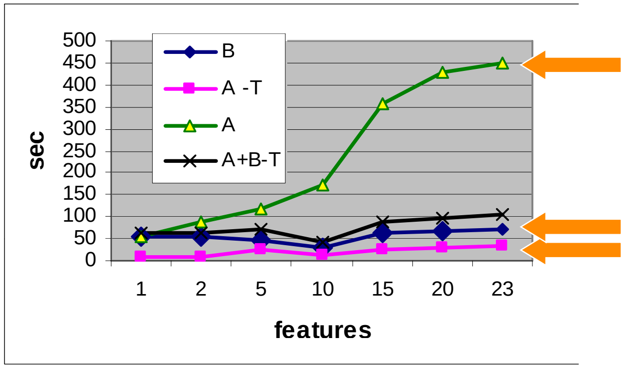
- Start at upper left compute nodes on lower right
- #1: refine models and then derive
- #2: derive representations and then refine
- #2 is faster by a factor of 2-3
- Diagrams tell us different ways in which programs can be synthesized



Benefit: Interesting Optimization

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

see ICSE 2007 paper by Trujillo et al.



Experience

- Our tools initially did not satisfy properties commuting diagrams
  - synthesizing via different paths yielded different results
  - exposed errors in our tools & specifications
- Significance of commuting diagrams
  - validity checks provide assurance on the correctness of our model abstractions, portlet specifications, and our tools
  - applies also to individual transformations (as they too have commuting diagrams)
  - win – better understanding, better model, better tools
  - reduce problem to its essence

In the Future

- Theory, methodology, tools of architectural metaprogramming use elementary ideas from

Category Theory

  - where homomorphisms, pushouts, commuting diagrams arise...
  - finding utility in relating software structures to mathematical structures
  - preliminary results are encouraging

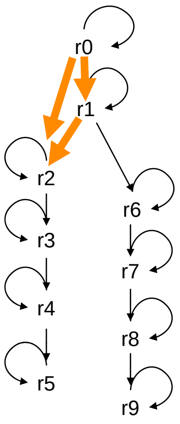
Conclusions

A Brief Tutorial on Category Theory

Conclusions

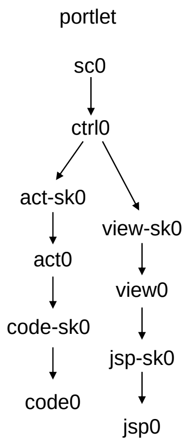
A Brief Tutorial on Category Theory

- **Category** is a directed graph with special properties
- Nodes are **objects**, edges are **arrows**
- Arrows are maps that compose
- Arrow composition is associative
- Identity arrows



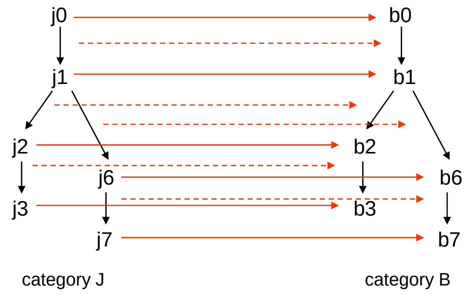
A Category – Look Familiar?

- **Category** is a directed graph with special properties
- Nodes are **objects**, edges are **arrows**
- Arrows are maps that compose
- Arrow composition is associative
- Identity arrows are implied



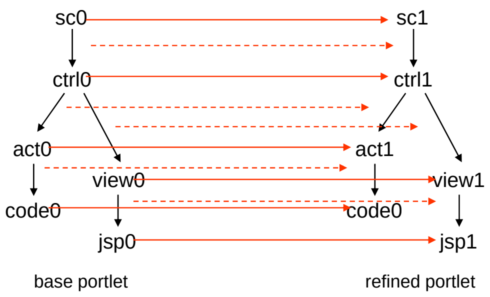
Functors

- Structure preserving map between 2 categories
  - embedding of category J into B such that J’s connectivity properties are preserved
- **Manifest functor** between isomorphic categories
  - map each object, arrow in J to the corresponding object, arrow in B



Functors – Look Familiar?

- Structure preserving map between 2 categories
  - embedding of category J into B such that J’s connectivity properties are preserved
- **Manifest functor** between isomorphic categories
  - map each object, arrow in J to the corresponding object, arrow in B


sc0
sc1
ctrl0
ctrl1
act0
act1
view0
view1
code0
code0
jsp0
jsp1
base portlet
refined portlet

features are functors

Next Steps

- Can express many of the ideas of architectural metaprogramming in terms of categorical concepts
- Much more to come...

Conclusions

Conclusions

- Extraordinarily good at:
  - languages
  - compilers
  - optimizations
  - analyses
- for programming in the **small** because we:
  - understand abstractions
  - their models
  - their relationships
  - their integration

- Not good at:
  - languages
  - compilers
  - optimizations
  - analyses
- programming in the **large** because we don’t fully:
  - understand abstractions
  - their models
  - their relationships
  - their integration

My Message: Getting Closer

- Fundamental ideas of metaprogramming
  - programs are values, transformations, operators
- Provide a simple explanation of technologies that are being developed and built in isolation – there is a lot in common with simple mathematical descriptions
- Recent work in program refactoring, synthesis, and model driven design are raising level of automation
  - success is not accidental
  - examples of paradigm called architectural metaprogramming that we are only now beginning to recognize
  - **many details and connections to other work are still not understood**

In the Future...

- Build tools, languages, and compilers to implement metaprogramming abstractions
  - improve structure of programs
  - higher-level languages & declarative languages
  - IDEs will be component evolution calculators
  - compilers will be program calculators
  - our understanding of programs, their representations, and their manipulation will be greatly expanded beyond source code
- Exciting future awaits us

Recommended Readings

- Ancona, Damiani, Drossopoulou. “Polymorphic Bytecode: Compositional Compilation for Java-like Languages”, *POPL 2005*.
- Batory. “Multi-Level Models in Model-Driven Development, Product-Lines, and Metaprogramming”, *IBM Systems Journal*, 45#3, 2006.
- Batory. “From Implementation to Theory in Product Synthesis”, *POPL 2007* keynote.
- Bezivin. “From Object Composition to Model Transformation with the MDA”, T*OOLS’USA* 2001.
- Binkley, et al. “Automated Refactoring of Object Oriented Code into Aspects”, *ICSM* 2005.
- Brown, Booch, Iyengar, Rumbaugh, Selic. “An MDA Manifesto”, Chapter 11 in *Model-Driven Architecture Straight from the Masters*, Frankel and Parodi, Editors, Meghan-Kiffer Press, 2004.
- Cole, Borba. “Deriving Refactorings for AspectJ”, *AOSD* 2005.
- Dig, Comertoglu, Marinov, Johnson. “Automated Detection of Refactorings in Evolving Components”, *ECOOP* 2006.
- Dig, Johnson. “How do APIs Evolve? A Story of Refactoring”, *Journal of Software Maintenance and Evolution*, 18#2, 2006.
- Hanenberg, et al. “Refactoring of Aspect-Oriented Software”. *Net.ObjectDays* 2003.

Recommended Readings

- Kleppe, Warmer, Bast. *MDA Explained: The Model-Driven Architecture -- Practice and Promise*, Addison-Wesley, 2003.
- Kulkarni, Reddy. “Model-Driven Development of Enterprise Applications”, in UML Modeling Languages and Applications, Springer LNCS 3297, 2005.
- Lopez-Herrejon, Batory, and Lengauer. “A Disciplined Approach to Aspect Composition”, *PEPM* 2006.
- Monteiro, Fernandes. “Towards a Catalog of Aspect-Oriented Refactorings”, *AOSD* 2005.
- Pierce. Basic Category Theory for Computer Scientists, MIT Press, 1991.
- Schmidt. “Model-Driven Engineering”. *IEEE Computer* 39(2), 2006.
- Smith. “A Generative Approach to Aspect Oriented Programming”, GPCE 2004.
- Sunyé, Pollet, Le Traon, Jézéquel. “Refactoring UML Models”. *Int Conf. UML*, 2001.
- Sztipanovits, Karsai. “Model Integrated Computing”, *IEEE Computer*, April 1997.
- Trujillo, Batory, Diaz. “Feature Oriented Model-Driven Development: A Case Study for Portlets”, *ICSE* 2007.
- Zhang, Lin, Gray. “Generic and Domain-Specific Model Refactoring using a Model Transformation Engine”, in *Model-driven Software Development*, Springer 2005.

Feature Interactions and Program Cubes

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

AHEAD
Feature
Oriented
Programming

Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy

Feature Interactions

- Are unavoidable
- Features interact by changing each others code or behavior
- This lecture looks at one fundamental form of feature interaction called
  Program Cubes (or Cubes)
  - there are other forms of interaction
- Formalized as tensors (multi-dimensional arrays)

A Micro Example

The Calculator Model

- Product line of calculators
  - what operations do you want in your calculator?

```
C = {
        Base,  // base program  }     constant

        Add,   // add           }
        Sub,   // subtraction   }     functions
        Form,  // format        }
        ...
}
```

Form●Add●Base

- How to express calculators with optional front-ends?
  - none, command-line, GUI$_1$, GUI$_2$, etc

Refactor Model C: Separate Core from GUI

$$\begin{array}{ccccc} \textbf{C} & & \textbf{Gui}_1 & & \textbf{Core} \\ \begin{bmatrix} \text{Base} \\ \dots \\ \text{Add} \\ \text{Sub} \end{bmatrix} & = & \begin{bmatrix} \text{Base} \\ \dots \\ \text{Add} \\ \text{Sub} \end{bmatrix} & \bullet & \begin{bmatrix} \text{Base} \\ \dots \\ \text{Add} \\ \text{Sub} \end{bmatrix} \end{array}$$

Model Synthesis

- We are synthesizing models!
- A **meta model** is a model of models
  - product-line of FOP models

$$MM = [\ Core, Gui_1, Gui_2, ..., Gui_n\ ]$$

- Features of MM are themselves base models or model refinements!

Core = [Base, Add, Sub, Form ...] // base calculator model

Gui_1 = [Base, Add, Sub, Form ...] // Gui_1 extensions to Core

Gui_n = [Base, Add, Sub, Form ...] // Gui_n extensions to Core

---

Model Synthesis

- To get desired model, compose Core with desired Gui

C = Gui_1 ● Core // original model

Desired = Gui_n ● Core // desired model

- To specify a calculator need pair of expressions
  - expression to produce a model
  - expression to produce a calculator
- **Vast symmetries are fundamental and common to FOP and program families**
- Now let's look at the mathematics behind all this

Tensors

Tensors

- n-dimensional arrays
- The **rank** of a tensor is the number of array indices required to describe it
  - cube is a 3D array (tensor of rank 3)
  - matrix is a 2D array (tensor of rank 2)
  - vector is a 1D array (tensor of rank 1)
  - scalar is a 0D array (tensor of rank 0)
- Number of elements along an index is its **dimension**
- Example: a rank 3 tensor of dimension (2,5,7) is a 3-dimensional array of size 2 × 5 × 7

Basic Tensor Concepts

Tensor Notation

- # of indices indicates rank of tensor
  - name of index is unimportant
- M_{ij} – tensor of rank 2
- M_{ijkn} – tensor of rank 4
- Scalar is rank 0

Tensor Product

- Cross product of elements of 2 tensors

$$T_{ij} \otimes S_{kn} = M_{ijkm}$$

- T is of rank t dim dt
- S is of rank s dim ds
- M is of rank t+s dim dt×ds

Tensor Product Example

- R_i = [A, B, C] tensor rank 1 dim 3
- S_k = [D, E, F, G] tensor rank 1 dim 4

$R_i \otimes S_k$ =

AD	AE	AF	AG
BD	BE	BF	BG
CD	CE	CF	CG

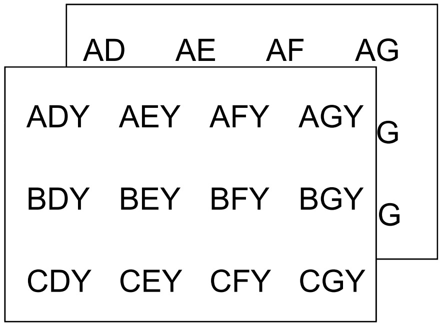
result is tensor of rank 2 = 1+1

dimension 3×4

Tensor Product Example

- T_l = [X, Y] tensor rank 1 dim 2

$(R_i \otimes S_k) \otimes T_l$ =



result is tensor of rank 3 = (1+1)+1

dimension (3×4)×2

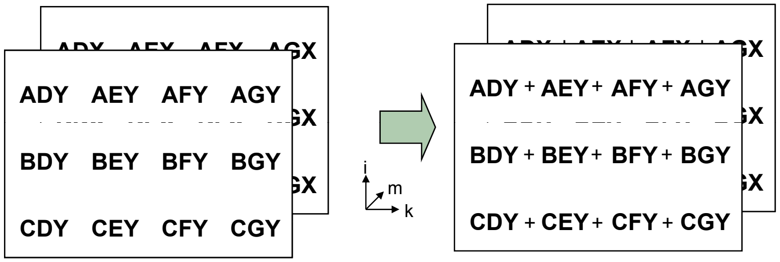
Tensor Contraction

- Aggregation of entries of a tensor reduces its rank
- Example: contracting k index of tensor T_{ikm} yields S_{im}

$$S_{im} = \Sigma_k T_{ikm}$$

rank 3
dim (3×4×2)

rank 2
dimension
(3×2)



ADY	AEY	AFY	AGY
BDY	BEY	BFY	BGY
CDY	CEY	CFY	CGY

ADY + AEY + AFY + AGY

BDY + BEY + BFY + BGY

CDY + CEY + CFY + CGY

Tensor Contraction

- **Order of aggregation does not matter!**

$$\text{scalar} = \Sigma_{ikm} T_{ikm}$$

$$= \Sigma_i \Sigma_k \Sigma_m T_{ikm}$$

$$= \Sigma_m \Sigma_i \Sigma_k T_{ikm}$$

$$= \ldots$$

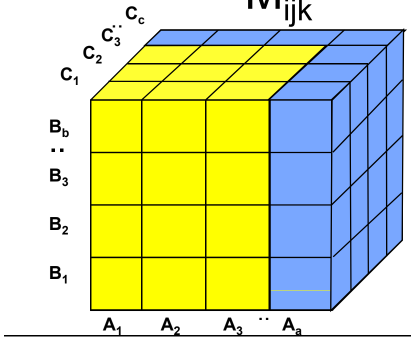
There are 3! different summation orders all yield the same scalar result

---

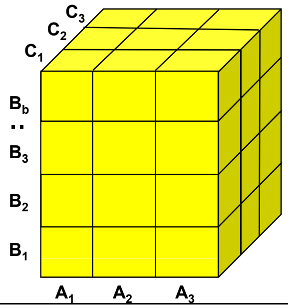
Tensor Projection

- Remove elements from dimensions
  - not a classical operation in tensor calculus
  - similar to data cubes of database systems

M_{ijk}



$\Pi_{i\in(1..3)\ k\in(1..3)} M_{ijk}$



Program Cubes

Program Cubes (PCs)

- Are a fundamental design technique in FOP
- Given model $F = [\ F_n, \dots\ F_2, F_1\ ]$ // notice vector
- Let program $G = F_8 + F_4 + F_2 + F_1$
  - where + denotes composition operator •
- Can write G as:

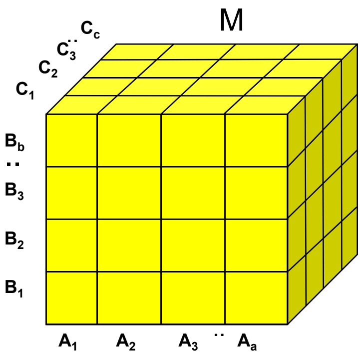
$$G = \Sigma_{i \in (8,4,2,1)} F_i$$

Generalize Interpretation

- An FOP model is a vector
  - $F = [\ F_n, \dots\ F_2, F_1\ ]$
  - no longer a set
  - tensor of rank 1, dimension n
- A program $G = \Sigma_{i \in (8,4,2,1)} F_i$
  - is a projection of model F that includes only the needed features
  - features in the vector are in composition order
  - vector is then contracted to a scalar

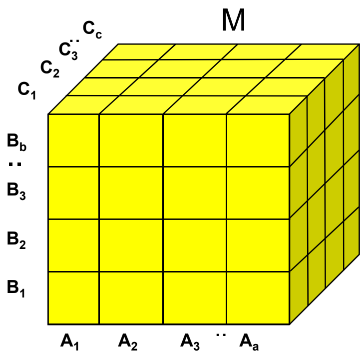
Program Cubes

- Use *n* rank-1 FOP models called **dimension models** to specify features or indices along a dimension
- A 3-D model M with A, B, C as dimension models
  - $A = [\ A_1, \dots\ A_a\ ]$
  - $B = [\ B_1, \dots\ B_b\ ]$
  - $C = [\ C_1, \dots\ C_c\ ]$



Program Cubes

- M is a tensor product: $A \otimes B \otimes C$
- M has a×b×c entries
- Entry M_{ijk} implements the **interaction of features** (A_i, B_j, C_k)
  - examples shortly



N-Dimensional Models

- A program is now specified by *n* expressions
  - 1 per dimension
- Program P in product-line of M has 3 expressions:

$$P = A_6 + A_3 + A_1 = \Sigma_{i\in(6,3,1)} A_i$$

$$P = B_7 + B_4 + B_3 + B_2 = \Sigma_{j\in(7,4,3,2)} B_j$$

$$P = C_9 + C_1 = \Sigma_{k\in(9,1)} C_k$$

Contracting Tensors

- The 3-expression specification of P is translated into an M expression scalar by contracting M along each dimension

$$P = \underbrace{\Sigma_{i\in(6,3,1)}}_{\text{A indices}} \underbrace{\Sigma_{j\in(7,4,3,2)}}_{\text{B indices}} \underbrace{\Sigma_{k\in(9,1)}}_{\text{C indices}} M_{ijk}$$

- Really a projection and contraction to a scalar:

$$P = \Sigma_{ijk} \left(\Pi_{i\in(6,3,1)} \Pi_{j\in(7,4,3,2)} \Pi_{k\in(9,1)} M_{ijk} \right)$$

Contracting Tensors

- **Order in which dimensions are summed (contracted) does not matter!**

$$P = \underbrace{\Sigma_{k\in(9,1)}}_{\text{C indices}} \underbrace{\Sigma_{j\in(7,4,3,2)}}_{\text{B indices}} \underbrace{\Sigma_{i\in(6,3,1)}}_{\text{A indices}} M_{i,j,k}$$

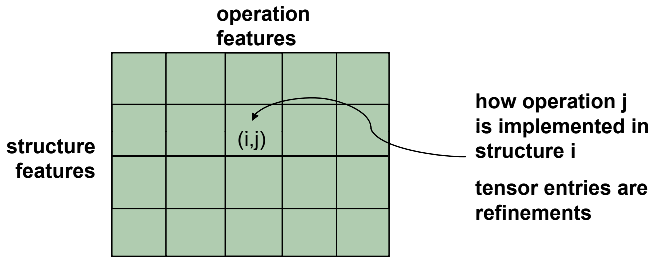
- Commutativity property of tensor contraction
- **Provided that dimensions are orthogonal**
  - **this needs to be proven**

Significance is Scalability!

- Complexity of program is # of features
- Given *n* dimensions with *d* features per dimension
  - program complexity is $O(d^n)$
  - using cubes $O(d\times n)$
  - ex: program P specified by $3\times4\times2$ features of M or only 3 + 4 + 2 dimensional features!
- **FOP program specifications are exponentially shorter when using cubes**

Academic Legacy

- “Extensibility Problem” or “Expression Problem”
  - classical problem in Programming Languages
  - see papers by: Cook, Reynolds, Wadler, Torgensen
  - focus is on achieving data type and operation extensibility in a type-safe manner



how operation j is implemented in structure i

tensor entries are refinements

Academic Legacy

- Multi-Dimensional Separation of Concerns (MDSoC)
  - Tarr, Ossher IBM

- Cubes are tensor formulation of MDSoC and Expression Problem
  - review a micro example (~35 line programs)
  - then a large example (~35K line programs) synthesis of the AHEAD Tool Suite
  - finally techniques to prove orthogonality of dimensions

Micro Example

Calculator Model revisited

Calculator Matrix

- View product-line as a matrix
- Tensor product of $\text{Calc}_r \otimes \text{GUI}_c = \text{CT}_{rc}$

GUI model

Calc model

	GUI_1	Cmd	GUI_2	...	Core
...	...	...	...	...	...
Form	Form_1	Form_c	Form_2	...	Form
Sub	Sub_1	Sub_c	Sub_2	...	Sub
Add	Add_1	Add_c	Add_2	...	Add
Base	Base_1	Base_c	Base_2	...	Base

Calculator Synthesis is Tensor Contraction

- Define which GUI features to compose
  - MyCalc = GUI$_1$ + Core
  - project and contract the matrix

	GUI$_1$	Cmd	GUI$_2$	Core
…	…	…	…	…
Form	Form$_1$	Form$_c$	Form$_2$	Form
Sub	Sub$_1$	Sub$_c$	Sub$_2$	Sub
Add	Add$_1$	Add$_c$	Add$_2$	Add
Base	Base$_1$	Base$_c$	Base$_2$	Base

Calculator Synthesis is Tensor Contraction

- Define which GUI features to compose
  - MyCalc = GUI$_1$ + Core
  - project and contract the matrix

	GUI$_1$	+	Core
…	…	+	…
Form	Form$_1$	+	Form
Sub	Sub$_1$	+	Sub
Add	Add$_1$	+	Add
Base	Base$_1$	+	Base

Calculator Synthesis is Tensor Contraction

- Define which Calc features to compose
  - MyCalc = Add + Base
  - project and contract the matrix

	GUI$_1$	+	Core
…	…	+	…
Form	Form$_1$	+	Form
Sub	Sub$_1$	+	Sub
Add	Add$_1$	+	Add
Base	Base$_1$	+	Base

Calculator Synthesis is Tensor Contraction

- Define which Calc features to compose
  - MyCalc = Add + Base
  - project and contract the matrix

$$\text{MyCalc} = \text{Add}_1 + \text{Add} + \text{Base}_1 + \text{Base}$$

process is symmetrical
get equivalent result if
rows are contracted
first

	GUI$_1$	+	Core	
Add	Add$_1$	+	Add	+
Base	Base$_1$	+	Base	

Calculator Synthesis is Tensor Contraction

- Define which Calc features to compose
  - MyCalc = Add + Base
  - project and contract the matrix

	GUI_1	Cmd	GUI_2	Core
...	...	...	...	...
Form	$Form_1$	$Form_c$	$Form_2$	Form
Sub	Sub_1	Sub_c	Sub_2	Sub
Add	Add_1	Add_c	Add_2	Add
Base	$Base_1$	$Base_c$	$Base_2$	Base

Calculator Synthesis is Tensor Contraction

- Define which Calc features to compose
  - MyCalc = Add + Base
  - project and contract the matrix

	GUI_1	Cmd	GUI_2	Core
Add + Base	Add_1 + $Base_1$	Add_c + $Base_c$	Add_2 + $Base_2$	Add + Base

---

Calculator Synthesis is Tensor Contraction

- Define which GUI features to compose
  - MyCalc = GUI_1 + Core
  - project and contract the matrix

	GUI_1	Cmd	GUI_2	Core
Add + Base	Add_1 + $Base_1$	Add_c + $Base_c$	Add_2 + $Base_2$	Add + Base

---

Calculator Synthesis is Tensor Contraction

- Define which GUI features to compose
  - MyCalc = GUI_1 + Core
  - project and contract the matrix

MyCalc = Add_1 + $Base_1$ + Add + Base

	GUI_1	+	Core
Add + Base	Add_1 + $Base_1$	+	Add + Base

Calculator Synthesis is Tensor Contraction

- Note generated expressions are not syntactically identical
  - columns, rows:

$$MyCalc = Add_1 + Add + Base_1 + Base$$

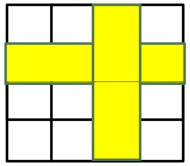
  - rows, columns:

$$MyCalc = Add_1 + Base_1 + Add + Base$$

- Expressions are equal because Add and $Base_1$ are commutative (orthogonal)
  - see how we prove this property later…

When to Use Multiple Dimensions?

- Rule: When adding a feature requires the lock-step updating of many other features
  - row feature updates all columns
  - column feature updates all row features



A Macro Example

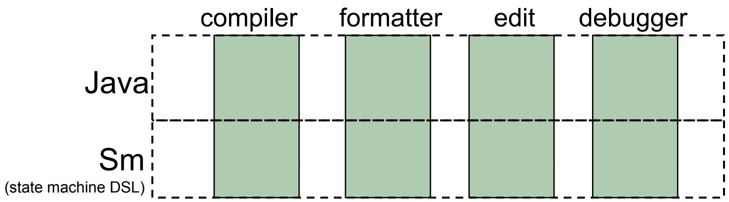
Synthesizing the AHEAD Tool Suite

Perspective

- So far, our models customize **individual programs**
  - set of all such programs is a **product-line**
- **Tool Suite** is an integrated set of programs, each with different capabilities
  - MS Office (Excel, Word, Access, ...)
- Question: Do features scale to tool suites?
  - product-line of tool suites

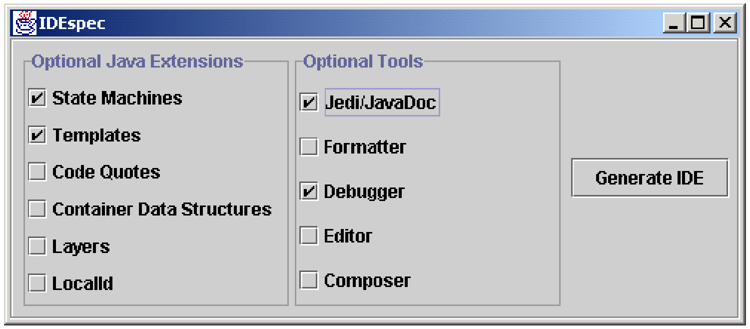
IDEs: A Tool Suite

- **Integrated Development Environment (IDE)**
  - suite of tools to write, debug, document programs
  - AHEAD variant: Java language extensibility



In principle, features scale!!!

The Problem – Declarative IDE



Optional Java Extensions
- ☑ State Machines
- ☑ Templates
- ☐ Code Quotes
- ☐ Container Data Structures
- ☐ Layers
- ☐ LocalId

Optional Tools
- ☑ Jedi/JavaDoc
- ☐ Formatter
- ☑ Debugger
- ☐ Editor
- ☐ Composer

From this declarative DSL spec, how do we generate AHEAD tools?

Define Dimensional Model #1

- AHEAD Model of Java Language Dialects

$$\texttt{J = [} \overbrace{\texttt{Java,}}^{\text{constant}} \; \overbrace{\texttt{Sm, Tmpl, Ds, ...}}^{\text{functions (optional features)}} \texttt{]}$$

- Dialects of Java specified by expression

```
Jak =  Tmpl + Sm + Java          // java +
                                 // state machines +
                                 // templates
…
```

Define Orthogonal Model #2

- Tools can be specified by a different, orthogonal model

$$\texttt{IDE = [} \overbrace{\texttt{Parse,}}^{\text{constant}} \; \overbrace{\texttt{ToJava, Harvest, Doclet, ...}}^{\text{functions (optional features)}} \texttt{]}$$

- Different tools have different expressions

```
jak2java =  ToJava + Parse

jedi     =  Doclet + Harvest + Parse
...
```

Tool Specification

- Defined by a pair of expressions
  - one defines tool language
  - other defines tool actions

  - ex: **`jedi`** (i.e., **`javadoc`**) for the Jak dialect of Java

```
jedi = Tmpl + Sm + Java                 // using J Model

jedi = Doclet + Harvest + Parse        // using IDE Model
```

- Synthesize **`jedi`** by projecting and contracting the tensor product of the J and IDE models

---

Tensor for `jedi`

- Rows are language features
- Columns are tool features
- Entries are modules (refinements) that implement a language feature for a tool feature
- Shows relationship between IDE and J models

	Doclet	Harvest	Parse
Java	JDoclet	JHarvest	JParse
Sm	SDoclet	SHarvest	SParse
Tmpl	TDoclet	THarvest	TParse

Cube for jedi

Tensor for `jedi`

- Composition of these modules yields **`jedi`**
- Synthesize **`jedi`** expression by contracting the tensor according to its dimensional expressions

	Doclet	Harvest	Parse
Java	JDoclet	JHarvest	JParse
Sm	SDoclet	SHarvest	SParse
Tmpl	TDoclet	THarvest	TParse

Tensor for jedi

Contract the Tensor!

- IDE expression

  `jedi = Doclet + Harvest + Parse`

- Tells us the column summation order

	Doclet	+	Harvest	+	Parse
Java	JDoclet	+	JHarvest	+	JParse
Sm	SDoclet	+	SHarvest	+	SParse
Tmpl	TDoclet	+	THarvest	+	TParse

Sum remaining columns

Now Contract the Rows

- J expression

`jedi = Tmpl + Sm + Java`

- Tells us the row summation order

	Doclet	+	Harvest	+	Parse
Java	JDoclet	+	JHarvest	+	JParse
+			+		
Sm	SDoclet	+	SHarvest	+	SParse
+			+		
Tmpl	TDoclet	+	THarvest	+	TParse

now add
Tmpl Row

Resulting Expression

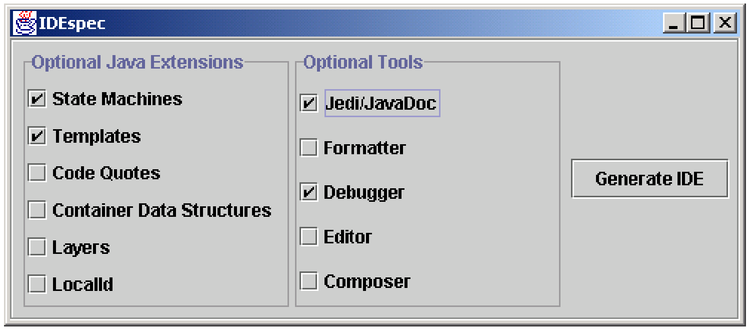
```
jedi = ( TDoclet + THarvest + TParse ) +
       ( SDoclet + SHarvest + SParse ) +
       ( JDoclet + JHarvest + JParse )
```

Using Cubes we can synthesize an expression for a language-dialect specific tool

---

Using Cubes to Generate

- Tool Suites...



Optional Java Extensions

☑ State Machines
☑ Templates
☐ Code Quotes
☐ Container Data Structures
☐ Layers
☐ LocalId

Optional Tools

☑ Jedi/JavaDoc
☐ Formatter
☑ Debugger
☐ Editor
☐ Composer

Product-Line Tensor

- That relates J and IDE models
- Rows are language features
- Columns are tool features
- Entries implement feature interactions (refinements)

	Parse	ToJava	Harvest	Doclet	Signat
Java	JParse	J2Java	JHarvest	JDoclet	JSig
Sm	SParse	S2Java	SHarvest	SDoclet	SSig
Tmpl	TParse	T2Java	THarvest	TDoclet	TSig
Ds	DParse	D2Java	DHarvest	DDoclet	DSig

To Synthesize IDE Tools

- Project unneeded rows and columns
  - directly from IDE GUI input
  - example: jedi, jak2java for Java + Sm + Tmpl

	Parse	ToJava	Harvest	Doclet
Java	JParse	J2Java	JHarvest	JDoclet
Sm	SParse	S2Java	SHarvest	SDoclet
Tmpl	TParse	T2Java	THarvest	TDoclet

Tensor for IDE Tools

- Contract rows
- Note the semantics of the result...

	Parse	ToJava	Harvest	Doclet
Java	JParse	J2Java	JHarvest	JDoclet
	+	+	+	+
Sm	SParse	S2Java	SHarvest	SDoclet
	+	+	+	+
Tmpl	TParse	T2Java	THarvest	TDoclet

Yields Expression For Each Tool Feature!

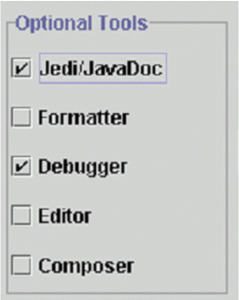
```
Parse     =  TParse    +   SParse    +   JParse
ToJava    =  T2Java    +   S2Java    +   J2Java
Harvest   =  THarvest  +   SHarvest  +   JHarvest
Doclet    =   TDoclet   +  SDoclet   +  JDoclet
```

- And we know expressions for each tool!

```
jak2java  = ToJava + Parse
jedi      = Doclet + Harvest + Parse
...
```

IDE Generator is Simple

- For each selected tool, evaluate its expression



Optional Tools

☑ Jedi/JavaDoc

☐ Formatter

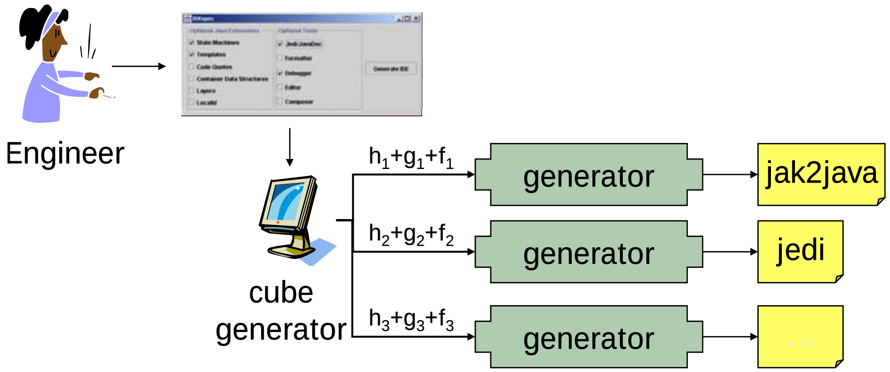
☑ Debugger

☐ Editor

☐ Composer

And generate the code for each tool automatically!

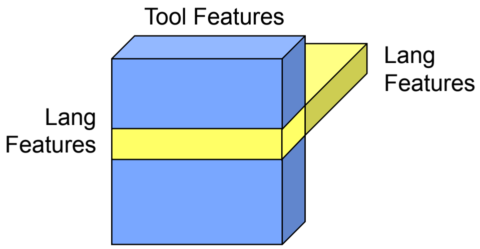
Generator of IDE Tool Suite



Experimental Results

Bootstrapping AHEAD

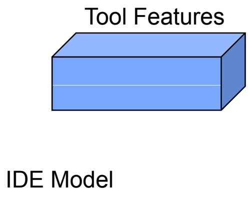
- We contracted a tensor of rank 3, dimension (8×6×8) to generate 5 tools of the AHEAD Tool Suite



3rd dimension captures language feature interactions

Bootstrapping AHEAD

- Contract tensor to produce IDE model, from which we can generate tool expression



Sum 3rd dimension
Sum rows

Results of AHEAD Bootstrap

- 90 distinct features
- Typical tool contains 20-30 features
    - most tools share 10 features
- Generated Java for each tool is ~35K LOC
- Generating well close to 150K from simple, declarative specifications
    - exactly what we want
- Making designs for multiple tools to conform to a tensor
    - controlling the complexity of tool suites

Tensor Representations Scale!!

- Micro example ~150 LOC total
- AHEAD example ~150K LOC total
- **3 orders of magnitude!**
- **Cubes apply to all levels of abstraction equally**
- **Cubes scale to *much* larger systems**

Proving Commutativity Properties of Tensors

On going work…

Contracting Tensors

- We assumed a basic property of tensors
- **Order in which dimensions are contracted does not matter**
    - commutativity property that we have to verify
- Cubes need not be orthogonal, as next example shows

Example of Non-Orthogonal Cube

- A non-orthogonal Cube

```
void run() {
  lit.print();
}
```

```
void run() {
  Super().run();
  lit.eval();
}
```

```
void run() {
  Super().run();
  add.print();
}
```

```
void run() {
  Super().run();
  add.eval();
}                   (a)
```

by rows first

```
void run() {
  lit.print();
  add.print();
  lit.eval();
  add.eval();
}            (b)
```

by columns first

```
void run() {
  lit.print();
  lit.eval();
  add.print();
  add.eval();
}            (c)
```

So What?

- Contract tensors differently to provide different views of software
    - viewing modules from language feature viewpoint or tool feature view point is occasionally useful
- Properties derived in one view (contraction), might not hold in other views
- Edits or code repairs performed in one view might not work correctly in other views
- Need consistent views!!
    - simple design changes can make a cube orthogonal

A Fix: An Orthogonal Cube

- An orthogonal cube and its contraction

```
void run() {pr(); }
void pr()
  { lit.print(); }
```

```
void run()
  { Super().run(); ev(); }
void ev() { lit.eval(); }
```

```
void pr() {
  Super().pr();
  add.print();
}
```

```
void ev() {
  Super().ev();
  add.eval();
}                        (a)
```

```
void run()
{ pr(); ev(); }

void pr() {
  lit.print();
  add.print();
}
void ev(){
  lit.eval();
  add.eval();
}              (b)
```

Properties to Preserve

- Same program must be synthesized when tensor dimensions are contracted in any order
- For a tensor A of rank 2:

$$\Sigma_i \Sigma_k A_{ik} = \Sigma_k \Sigma_i A_{ik}$$

- For a tensor of rank n, there are n! summation orders, all must produce equivalent results
- Need algorithms to verify these properties

Orthogonality Property

- Reduces to testing 2D matrix

$$\begin{array}{|c|c|} \hline a_{11} & \boxed{a_{12}} \\ \hline \boxed{a_{21}} & a_{22} \\ \hline \end{array}$$

$$\Sigma_{i\in(1,2)}\Sigma_{j\in(1,2)}\ A_{ij} = \Sigma_{j\in(1,2)}\Sigma_{i\in(1,2)}\ A_{ij}$$

$$a_{11} + \boxed{a_{21} + a_{12}} + a_{22} = a_{11} + \boxed{a_{12} + a_{21}} + a_{22}$$

- For the above to be equal, the following must hold

$$a_{21} + a_{12} = a_{12} + a_{21}$$

  - composition of the bottom left and upper right quadrants must commute

---

a_{21} and a_{12} commute if

- (1) they do not add or refine the same member
  - they add or refine non-overlapping sets of methods and variables
- (2) they do not refer to members added by each other
- Both conditions are easy to verify;
the hard part is doing so efficiently
  - brute force doesn't work as it would be hideously slow

Essence of the Algorithm

- For an arbitrary rank, dimension tensor T
- For every member m added or refined in feature F, store it along with the coordinates of F in T in a hash table
- If a prior definition of m exists (meaning it was added or refined by another feature G), see if the coordinates of F and G conflict and if they do, see if F and G can belong in the same product
  - if so, T is not orthogonal
- Similar analysis for references
- Almost linear in the size of the code base

Example: Bali Tools of ATS

TOOLS

		Base	**CodGen**	**Bali2Jak**	**Bali2Javacc**	**Bali Composer**	**Bali2Layer**
L A N G.	**Core**	kernel, bali, visitor, collect	codegen	bali2jak	bali2javacc	composer ★	bali2layer, b2lOptns, b2lGUI
	WithReqFeature	require ★		reqComposer	reqB2Javacc	reqComposer	
	Without						

feature model grammar

```
Bali : Lang Tools ;

Lang : core CoreFeatures ;
CoreFeatures : withRequireFeature
             | without;

Tools: base [codeGen] Tool :: BaliTools;
Tool : bali2jak :: Bali2JakTool
     | bali2javacc :: Bali2javaccTool
     | baliComposer :: BaliComposerTool
     | bali2layer :: Bali2layerTool;

%%
BaliComposerTool implies not codeGen;
Bali2JakTool or Bali2layerTool or Bali2javaccTool iff codeGen;
```

require refines method defined in composer | bali2jak | ...

Example Error

- Require refines method defined in Composer

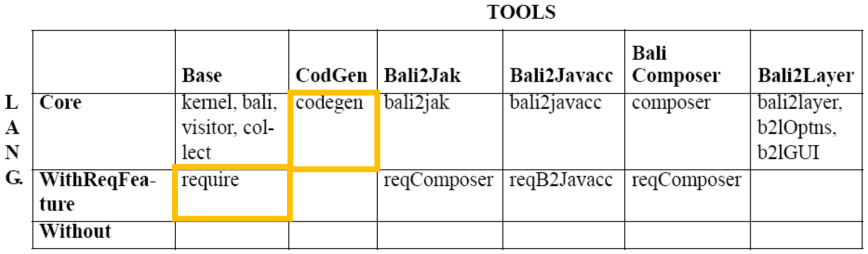
composer.main

```
public Object driver( String[] args ) throws Throwable {
        setVersion( "v2003.02.17" ) ;
        ...
        Collector collector = collectSources( inpFiles ) ;
       ...
        return collector ;
}
```

require.main

```
public Object driver( String args[] ) throws Throwable {
        setVersion( "v2002.09.03" ) ;
        return Super( String[] ).driver( args ) ;
}
```

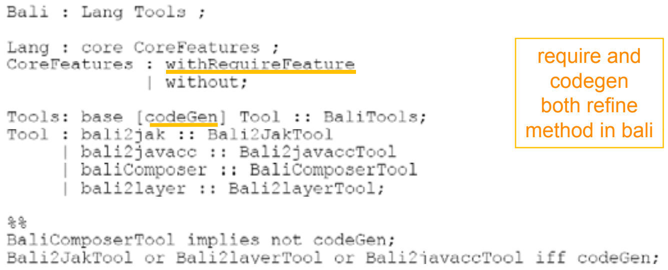
Another Error



TOOLS

LANG		Base	CodGen	Bali2Jak	Bali2Javacc	Bali Composer	Bali2Layer
L A N	**Core**	kernel, bali, visitor, collect	codegen	bali2jak	bali2javacc	composer	bali2layer, b2lOptns, b2lGUI
G.	**WithReqFeature**	require		reqComposer	reqB2Javacc	reqComposer	
	Without						

grammar



```
Bali : Lang Tools ;

Lang : core CoreFeatures ;
CoreFeatures : withRequireFeature
             | without;

Tools: base [codeGen] Tool :: BaliTools;
Tool : bali2jak :: Bali2JakTool
     | bali2javacc :: Bali2javaccTool
     | baliComposer :: BaliComposerTool
     | bali2layer :: Bali2layerTool;

%%
BaliComposerTool implies not codeGen;
Bali2JakTool or Bali2layerTool or Bali2javaccTool iff codeGen;
```

require and codegen both refine method in bali

Example Error

- Require and Codegen both refine method in Bali

bali.main

```
public Object driver( String args[] ) throws Throwable {
        ...
        return parseTree ;
}
```

codegen.main

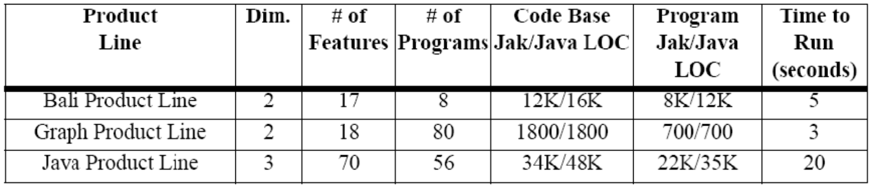
```
public Object driver( String args[] ) throws Throwable {
        setVersion( "v2002.09.04" ) ;
        return Super( String[] ).driver( args ) ;
    }
```

require.main

```
public Object driver( String args[] ) throws Throwable {
        setVersion( "v2002.09.03" ) ;
        return Super( String[] ).driver( args ) ;
}
```

Other Statistics

- Fast – didn't find errors in JPL



Product Line	Dim.	# of Features	# of Programs	Code Base Jak/Java LOC	Program Jak/Java LOC	Time to Run (seconds)
Bali Product Line	2	17	8	12K/16K	8K/12K	5
Graph Product Line	2	18	80	1800/1800	700/700	3
Java Product Line	3	70	56	34K/48K	22K/35K	20

Insights

- Oddly, we didn’t find serious errors in the ATS designs
  - only benign (inconsequential) errors were found
- Created these designs long before we had any analysis tools
  - suggests that creating orthogonal tensors is not difficult

---

Final Comments

Future Work

- Commutativity or “orthogonal” properties have a simple description in category theory
  - deep interconnection with our use of tensors
- Other forms of feature interactions
  - generalization of the ideas presented here seem to account for many of such interactions
  - developing theories and supporting tools for this
- Additional analyses
  - want to analyze product-lines to ensure that all legal compositions of features yield type safe programs
  - Thaker, Batory, Kitchin, Cook.
    “Safe Composition of Product Lines”, GPCE 2007

Recommended Readings

- Batory, Lopez-Herrejon, Martin, “Generating Product-Lines of Product Families”, Automated Software Engineering 2002.
- Batory, Liu, Sarvela, “Refinements and Multi-Dimensional Separation of Concerns”, ACM Sigsoft 2003.
- M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec, “Feature Interaction: A Critical Review and Considered Forecast”.Computer Networks, January 2003.
- Cook “Object-Oriented Programming versus Abstract Data Types”. Workshop on Foundations of Object-Oriented Languages, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178
- Harrison and Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, *OOPSLA 1993*, 411-427.
- Kay, “Tensor Calculus”, Shaums Outlines, 1988.
- J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications", ICSE 2006.

Recommended Readings

- Ossher and Tarr, “Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software.” CACM 44(10): 43-50, *October 2001*.
- Reynolds “User-defined types and procedural data as complementary approaches to data abstraction”. Reprinted in C.A. Gunter and J.C.Mitchell, Theoretical Aspects of Object-Oriented Programming, MIT Press,1994.
- Thaker, “Design and Analysis of MultiDimensional Program Structures”, M.Sc. Thesis, Dept. Computer Sciences, University of Texas at Austin, 2006.
- Thaker, Batory, Kitchin, Cook, “Towards Safe Composition of Product-Lines”, GPCE 2007.
- Tarr, Ossher, Harrison, and Sutton, “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, *ICSE 1999*.
- Torgensen “The Expresion Problem Revisited. “Four new solutions using generics”, ECOOP 2004.
- Wadler “The expression problem”. Posted on the Java Genericity mailing list (1998)