Design by Transformation (D×T)
Principles of Model Driven Engineering

Don Batory
Department of Computer Science
University of Texas at Austin
batory@cs.utexas.edu
Introduction

- My research is at intersection of
  - software product-lines (SPLs)
  - program refactoring
  - model driven engineering (MDE)
  - database systems

- I come from world of
  - informal software engineering and design
  - not compilers, formal software development, mathematics

- What distinguishes my work
  - start with practice, find a theory that fits practice
  - I use algebra to explain my ideas
  - foundation for more formal theories of automated development
Keys to the Future

• New paradigms will 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

• The most significant result in automated program design and development, period

• Not mentioned in typical SE texts …
Relational Query Optimization (RQO)

- Declarative query is mapped to an relational algebra expression
- Each expression represents a unique program
- Expression is optimized using algebraic identities
- Efficient program generated from expression

SQL select statement

parser

inefficient relational algebra expression

optimizer

efficient relational algebra expression

code generator

declarative domain-specific language

automatic programming

generative programming

efficient program
Keys to Success

- Automated development of query evaluation programs
  - hard-to-write, hard-to-optimize, hard-to-maintain
  - revolutionized and simplified database usage

- Represented program designs as algebraic expressions
  - different expressions represented different programs
  - compositions of relational operations

- Compositionality is hallmark of great engineering

- Use algebraic identities to optimize expressions
  - equates the semantics of different programs
This Tutorial

• Sketch the future:
  • automated software design & maintenance from an algebraic perspective
  • extrapolating 25+ years of experience
  • characteristics of new languages, compilers, tools

• Essential complexity of software structure
  • is exposed when program construction and design is viewed as a computation
  • hides accidental complexity

• Design by Transformation (D×T)
  – generalization of RQO paradigm
  • programs are values
  • transformations map programs to programs
  • operators map transformations to transformations
  • meta-expressions
Tutorial Overview

1. Program Refactoring, Program Synthesis, and Model Driven Engineering
   algebra underlies program construction

2. Objects and Arrows of $\mathbf{D} \times \mathbf{T}$
   the relevance of categories and category theory

3. Extraction of MDE Architectures from Parallel Streaming Applications
   stepwise development of software architectures by applying domain-specific identities

Buckle Up!
Lecture 1:
Program Refactoring, Program Synthesis, and Model Driven Engineering
Upcoming Topics – Four Mini Talks

1. Basics of $D \times T$

2. Program Refactoring
   - Dig & Johnson (Illinois)

3. Program Synthesis
   - Lopez-Herrejon (Texas) & Lengauer (Passau)

4. Model Driven Engineering
   - Trujillo & Diaz (Basque Country)

- All describe systems that have been built
  - step back and give a simple $D \times T$ explanation of their results
  - pave way for following lectures
#1: Basics of D×T

Design by Transformation
Personal Edition

scalable solutions for mission-critical designs
• Programs are **values**

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

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

• Here is another value:

```java
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 $C_1 =$
  
  ```java
  class C {
    void comp () {..}
  }
  ```
  
- Let $C_2 =$
  
  ```java
  class C {
    int x;
    void inc() {...}
  }
  ```

- $C_1 + C_2 =$
  
  ```java
  class C {
    void comp () {..}
    int x;
    void inc() {...}
  }
  ```
+ (Sum) is Disjoint 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)
    \[ (A + B) + C = A + (B + C) \]

- So what? Why are these properties important?
2\textsuperscript{nd} Operation: – (Subtraction)

- Subtraction is set difference

\[(D + C) – C = D\]

- Has expected properties:
  - left associative \[P – C – D = ((P – C) – D)\]
  - not commutative \[P – C \neq C – P\]
  - identity \[P – 0 = P\]

- Again, we need these rules why?
3\textsuperscript{rd} Operation: Distributive Transformations

- **Transformation** is a function that maps a program to another program.

- **Rename**(\(p, q, r\)) – in program “\(p\)” replace name “\(q\)” with “\(r\)”

\[
\text{Rename(}
\begin{array}{c}
\text{class C} \\
\text{i} \text{n} x; \\
\text{v}o\text{i}d \text{ i}n\text{c}() \{ .x. \} \\
\ldots \\
\} \\
\end{array}
\), \(C.x, C.z\) =
\begin{array}{c}
\text{class C} \\
\text{i} \text{n} z; \\
\text{v}o\text{i}d \text{ i}n\text{c}() \{ . z . \} \\
\ldots \\
\} \\
\end{array}
\]
Another Example

- Called a **fixed point**: a value \( x \) such that \( f(x) = x \)

- Distributive transformations have lots of fixed points
Key Property (where they get their name)

- Transformations **distribute** over + and –

\[
f( A + B ) = f(A) + f(B) \\
f( C - D ) = f(C) - f(D)
\]

- Here’s an example...
Example of Distributivity

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

\[\text{Rename}(\text{class } D, C.x, C.z)\]

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

\[\text{Rename}(\text{class } C, C.x, C.z)\] \[=\]

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

\[\text{Rename}(\text{class } D, C.x, C.z)\]

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

\[\text{Rename}(\text{class } C, C.x, C.z)\]

\[+\]

Basics- 12
Structures & Properties

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

  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  = 6*E^2  // E is edge length
  - volume  = E^3
Design

- **Design** is a meta-expression (or more generally a meta-program) **that** says how to construct a program (structure)
Note!

- Many meta-expressions produce the same program

- This means there are many ways in which a program can be designed and built

- Makes intuitive sense
Properties

- Property of a program – is derived from its structure

  - compilers verify type correctness of programs
    (in addition to translating program to bytecodes)

  - other research guarantees other properties (ex. security)
    of programs – also enforced by special compilers

  - but it is possible to write programs that do not have the
    properties we want – why we write tests

In this lecture,
I focus on program structure.

The above list gives you an idea of common properties to check.
#2: Advances in Program Refactoring
Refactoring

- Is a transformation that changes the structure of a program, but not the property of its behavior
  - rename methods
  - move method from subclass to superclass ...

- Most design patterns are end-products of refactorings
  - see Kerievsky, "Refactoring to Patterns“ text

- Common IDEs (Eclipse, Visual Studio, IntelliJ) have refactoring tools or plug-ins

- Here’s an interesting refactoring problem noticed by Dig and Johnson ~2005
  - resulted in an addition to Eclipse in 2007
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
  - instance method becomes static method of host class
  - moved method takes instance of home class as extra argument
  - references to old method replaced with calls to moved method

Component

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

```java
class home {
    X m(..) { ... }
    void b() { host.m(.., f) }
}
```

Client Code

```java
class bar {
    void y() {
        host.m(.., f)
    }
}
```
A Common API Change

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

But a component developer doesn’t have the client code!!
User must make changes manually!

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

class home {
    void b() { host.m(..,f) }
}

class bar {
    void y() {
        host.m(..,f)
    }
}

Component  Client Code
Express Change as a Meta-Expression

\[ P_{\text{new}} = \rho \cdot \mu \left( P_{\text{old}} \right) \]

Component

Client Code

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

class home {
    X m(.) { ... }
    void b() \cdot host.m(., f)
}

class bar {
    void y() {
        host.m(., f)
    }
}
Other Common API Changes

- Move Field

- Delete Method
  - usually done after method is renamed or moved

- Change Argument Type
  - ex: replace argument type with its supertype

- Factory Method
  - add a factory method to a class

- Lots of others...
  - preliminary work suggests all can be written as meta-expressions
• Manually analyzed change logs and documentation 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 meta-expressions
In the Future

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

\[ \beta_3 \cdot e_3 \cdot e_2 \cdot \beta_2 \cdot \beta_1 \cdot e_1 \]

\[ \beta = \beta_3 \cdot \beta_2 \cdot \beta_1 \]

- API updates are expressed by $\beta$, a projection of changes where “private” edits are removed and only API changes remain

\[ \beta_3 \cdot e_3 \cdot e_2 \cdot \beta_2 \cdot \beta_1 \cdot e_1 = \text{version 1} \]

transformation to be applied to update client code w.r.t. changes in API
Client Update is a Meta-Function $U$

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

**Explanation:**

This is not the original presentation of the result; it is a D\times T explanation.
IDEs create update meta-functions like $U$ to distribute
  • transformations are distributed, not components

In Eclipse Since 2007

• IDEs transform code bases to automatically update them
  • $D \times T$ expresses the core idea elegantly
#3: Advances in Program Synthesis
Background

• Maintaining programs automatically via refactorings is useful

• But we also want to build programs automatically too

• We want simple declarative languages to specify programs
  • that any one could use
  • *not* starting with formal logic specifications

• Ans: **Software Product Lines**
  • *feature* is an increment in product functionality
  • features are a declarative way to specify customized programs
Features used in Engineering

- Dell web pages
  - declarative DSL
  - specify target product by features

- Other examples
  - design your own BMW
  - faucets, sinks
Declarative Program Specifications

- Graph Product Line

Program = Cycle \cdot Number \cdot DFS \cdot Weighted \cdot Directed
Scalability: Long History of Applications

- 1986 database systems 75K LOC
- 1989 network protocols
- 1993 data structures
- 1994 avionics
- 1997 extensible Java precompilers 35K LOC
- 1998 radio ergonomics
- 2000 program verification tools
- 2001 verified compiler for Java1.0
- 2002 fire support simulators
- 2003 AHEAD tool suite 250K LOC
- 2004 robotics controllers
- 2006 web portlets
- 2008 SGI+JavaScript application
- 2009 ZipMe compression library
Feature Oriented Programming (FOP)

- FOP is an example of $D \times T$: features are transformations

- Constants (*constant functions or values*)
  - $f$ – base program with feature $f$
  - $h$ – base program with feature $h$

- Unary Functions (*optional features*)
  - $i \cdot x$ – adds feature $i$ to program $x$
  - $j \cdot x$ – adds feature $j$ to program $x$

- An FOP “model” of a domain is an algebra
  \[ M = \{ f, h, \ldots, i, j, \ldots \} \]

- Different meta-expressions represent different programs of a product line
  \[ P_1 = i \cdot f \quad // \ P_1 \text{ has } i, f \]
  \[ P_2 = j \cdot i \cdot h \quad // \ P_2 \text{ has } j, i, h \]
FOP Implementations

- Use superposition
  - simpler (and more practical) than AOP
  - explore it in more detail in Lecture 2
  - basis for Bracha’s most recent language, Newspeak

- If we peer inside FOP features we see familiar ideas popularized by Aspect Oriented Programming (AOP)
  - here I use terminology of AOP, not AOP definitions
    - google “functional aspects”
  - ideas I use appeared long before AOP
Two Ideas

• **Introduction** – adds new members to existing classes
  • metaprogramming addition

• **Advice** – modifies methods at particular points, called **join points**
  • distributive transformation
  • advice is generally behavior-extending, *not* behavior-preserving

• No “subtraction” in AOP or in FOP
Introductions

- Incrementally add new members, classes

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

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

\[ P = C.b + C.\text{foo} + C.i + D.\text{bar} + D.\text{cnt} \]

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 point event occurs, execute a given piece of code

- Although advice has a “dynamic” interpretation, we can give it a “static” metaprogramming interpretation
  - **join point shadows**
  - how aspect compilers work

- View advice as a distributive transformation
  - when you advise a program, you advise all of its parts
class C {
    int i,j;
    void setI (int x){ i=x; }
    void setJ (int x){ j=x; }
}

after(): execution (void C.set*(..))
    { print("hi"); }
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 = hi(C.i + C.j + C.setI + C.setJ)
= hi(C.i) + hi(C.j) + hi(C.setI) + hi(C.setJ)
= C.i + C.j + hi(C.setI) + hi(C.setJ)
= C.i + C.j + C.setI' + C.setJ'
Features

- Features are transformations that:
  - introduce new terms \( (i) \)
  - advise or alter \( (\alpha) \) existing program \( (x) \)

\[
F(x) = i_f + \alpha_f(x)
\]

- Composition:

\[
G(F(B)) = i_g + \alpha_g(i_f + \alpha_f(b))
\]
In the Future

• Program designs will be calculations

• Compilers will be **program calculators**
  • inhale source code
  • generate meta-expression, maybe optimize expression
  • evaluate to synthesize program

• $D \times T$ expresses the core idea elegantly
  • google “AHEAD”, “FeatureHouse”

• 3\(^{rd}\) Lecture…
An Interesting Question:
What is the Relationship Between Advice and Refactorings?
Refactorings and advice are both transformations.

Suppose we have a refactoring and advice to apply to a program. What does it mean to compose them?

Note: Advice does not modify a refactoring. A refactoring is not a language construct; there are no join points in a refactoring.

Note: But a refactoring can modify programs with advice.
Example

Program P

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

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

change method names

change advice declaration
• Remember differential operators in calculus? 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

• Operators (expr to expr maps) is an example of:

  mapping of expressions of one algebra to expressions of another

• Grounded in Category Theory

  • theory of mathematical structures and their relationships
  • more later...
How Meta-Calculation Proceeds

Program P

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

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

\[ \beta \left( \text{hi} \left( C.i + C.j + C.SetI + C.SetJ \right) \right) \]
\[ = \beta \left( \text{hi} \right) \left( \beta(C.i) + \beta(C.j) + \beta(C.SetI) + \beta(C.SetJ) \right) \]
\[ = \beta \left( \text{hi} \right) \left( C.i + C.j + \beta(C.SetI) + \beta(C.SetJ) \right) \]
\[ = \beta \left( \text{hi} \right) \left( C.i + C.j + C.SetI + C.SetJ \right) \]
\[ = \text{HI} \left( C.i + C.j + C.SetI + C.SetJ \right) \]
Refactorings are operators on expressions that have higher-precedence than advice in $D \times T$

- operator maps a transformation to another transformation

Note:
- refactorings
- advice
- introductions
- modify structure of code
- but could also modify structure of grammars, makefiles, xml documents... as well

Algebraic viewpoint is universal – it applies to non-code representations as well

leads to our next topic…
#4: Advances in Model Driven Engineering
Model Driven Engineering (MDE) is a 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)
Metaprogramming Connection

• MDE embraces concept that program development is a computation
  – **claim**: MDE is a metaprogramming & D×T paradigm
  – models are values
  – transformations map models to models

• Common example
  
  java source
  
  javac
  
  class files

  *javadoc transforms java source to class files*
Interesting Question

- If `javac` is a transformation, is it distributive?

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

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

```
javac(                              )  
javac(                              ) +
```

```
classfile for D

classfile for C
```

Basics- 53
Interesting Question

javac is not distributive!

Although there is research by Ancona et. al. on Separate Class Compilation that makes it so…
More Typical MDE Example: PinkCreek

- Work with S. Trujillo and O. Diaz

- Portlet is a web component

- PinkCreek is an MDE case study for synthesizing portlets

- Uses transformations to map an annotated state chart (sc) to different representations (Java code, JSP code)
Portlet Synthesis Metaprogram

Example of using transformations to derive different models or representations of a program
Another Interesting Question...

As FOP and MDE are both metaprogramming paradigms, how do they combine?
Now and in the Future

• Features “extend” models

• An example:

\[ F(x) = i + \alpha(x) \]

\[ F'(x) = i' + \alpha'(x) \]
Fundamental Relationship

- Relationship between transformations that derive models from those that extend models

\[ D \circ F = F' \circ D \]

**Pushout**

**Commuting Diagram**
• Diagrams can be “pasted” together
• Given model in upper left, want to compute model in lower right
• Each path represents a different metaprogram
• Every path from upper left to lower right produces same result
Extending State Charts in PinkCreek

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

Base
- sView
- s1Search
- s2Select
- s3Summary
- s4Reserve
- s5Itinerary

Seat • Base
- sView
- s1Search
- s2Select
- s3Summary
- s4Reserve
- s5Itinerary
- s6Seating
Extending State Charts in PinkCreek

```plaintext
T"mk\(e\)(Δ\(S\)c, Δ\(F\)act-usr, Δ\(F\)view-usr, Δ\(F\)code-usr, Δ\(F\)jsp-usr)
{
  Δ\(F\)act\(_1\) = \(T'\) ctrl2act (Δ\(F\)act);
  Δ\(F\)act-sk\(_1\) = \(T'\) ctrl2act (Δ\(F\)act);
  Δ\(F\)view-sk\(_1\) = \(T'\) ctrl2view (Δ\(F\)ctrl);
  Δ\(F\)view = Δ\(F\)view-usr * Δ\(F\)view-sk;
  Δ\(F\)jsp-sk = Δ\(F\)act2jsp (Δ\(F\)view);
  Δ\(F\)jsp = Δ\(F\)jsp-usr * Δ\(F\)jsp-sk;
  Δ\(F\)raw = { Δ\(F\)ctrl, Δ\(F\)act, Δ\(F\)view, Δ\(F\)jsp, Δ\(F\)code, Δ\(F\)jspcode }; return Δ\(F\)raw;
}
```
Commuting Diagrams in PinkCreek

- Features map space of artifacts by extending them.

- Composing features sweeps out the commuting diagram to traverse to synthesize portlet representations.
Portlet Synthesis

• Shortest path is a **geodesic**

• Start at upper left compute nodes on lower right

• #1: extend models and then derive

• #2: derive representations and then extend

• #2 is faster than #1 by factor of 2-3 times

see ICSE 2007 paper by Trujillo et al.
Experience

- Our tools initially did not satisfy properties commuting diagrams
  - synthesizing via different paths yields different results
  - exposed errors in our tools & specifications

- Significance of commuting diagrams
  - validity checks provides assurances 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

• Better understanding of these ideas & their practicality

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

Category Theory

• where homomorphisms, pushouts, and commuting diagrams arise...

• topic of Lecture #2
Conclusions
We are…

• Extraordinarily good at:
  • languages
  • compilers
  • optimizations
  • analyses

• Programming in the **small**:
  • understand abstractions
  • their models
  • their relationships
  • their integration

• Not good at:
  • languages
  • compilers
  • optimizations
  • analyses

• Programming in the **large**:
  • ¬ understand abstractions
  • ¬ their models
  • ¬ their relationships
  • ¬ their integration
My Message: Getting Closer

- Program design and synthesis has a simple algebraic underpinning
  
  design is all about
  structure definition and manipulation
  
  which is what mathematics is about

- This lecture sets the stage for our next lectures
  
  category theory
  
  designs by algebra
Lecture 2: The Objects and Arrows of D×T

work with Maider Azanza and João Saraiva
Recap from Lecture 1

• Future of software design and development is automation
  • mechanize repetitive tasks
  • free programmers for more creative activities

• **Design by Transformation** is a paradigm where program design and synthesis is a computation

• **Design**: steps to take to create an artifact
  • meta-expression

• **Synthesis**: evaluate steps to produce the artifact
  • meta-expression evaluation

• **Design Optimization**: meta-expression optimization
Forefront of Automated Development

• **Model Driven Engineering (MDE)**
  - general-purpose approach
  - high-level models define applications
  - transformed into lower-level models

• **Software Product Lines (SPL)**
  - domain-specific approach
  - we know the problems, solutions of a domain
  - we want to automate the construction of these programs

• Both complement each other
  - strength of MDE is weakness of SPLs, and vice versa
  - not disjoint, but I will present their strengths as such here
This Lecture

• This is a modeling talk aimed at practitioners
  • no special mathematical background

• Review core ideas in **Category Theory**
  • theory of mathematical structures
  • result of a domain analysis of geometry, topology, algebra…
  • these concepts are fundamental to MDE, SPL
This Lecture

- Show categories provide unifying foundation for MDE & SPLs
- Series of mini-tutorials (10 minutes apiece)
#1: Categories in MDE

let’s start with some unfortunate terminology…
Objects in Categories

- **An object** is a domain of **points** (no standard term)
- **Metamodel** defines a domain of **models**
Examples

- MDE focuses on UML metamodels and their instances
- Ideas of objects & instances also apply to non-MDE artifacts
  - technical spaces of Bezivin, et al.

\[ j_1, j_2, j_3, j_4, j_5 \]

\[ x_1, x_2, x_3, x_4, x_5 \]

\[ b_1, b_2, b_3, b_4 \]
Recursion

• A point can be an object

• Standard MOF architecture

![Diagram showing recursion with objects and meta-metamodels](image)
Arrow

- Is a map or function or transformation or morphism between objects (all names are used)
  - implementation is unspecified
My Terminology (for this lecture)

- **Arrow** – denotes a map

- **Transformation** – an MDE implementation of an arrow
  - ATL, RubyTL, GReAT, QVT …

- **Tool** – is a non-MDE implementation of an arrow
  - standard tools of software engineers
• **Category** – a collection of objects and arrows

  • above is a category of 4 objects, 3 non-identity arrows
  
  • 4 identity arrows (not always shown)

  • categories satisfy 3 simple properties…
Properties of Categories

• Arrows are composable:

\[ A \circ (B \circ C) = (A \circ B) \circ C \]

• Composition is associative:  

• Identities

\[ F \circ \text{Id}_B = F \]

\[ \text{Id}_A \circ F = F \]
External Diagrams in MDE

- No standard names for such diagrams in MDE
  - drawn differently (without identity arrows)
  - Toolchain diagrams (MIC)
  - MegaModels (ATL)

- MDE “designs” are categories on an industrial scale
  - not the microscopic and often obscure examples in texts
External Diagrams in MDE

Legend:
- "-" - JRules operation
- ↓ - Injection
- → - Transformation
- ← - Extraction
- File
- Terminal model

provided by J. Bezivin
No standard name for such diagrams in MDE

Toolchain diagrams (MIC)

MegaModels (ATL)

External Diagrams in MDE

MSC

SC

M2MX

Java

M2TX

ByteCode

provided by J. Bezivin
Arrows with Multiple Inputs, Outputs

• Arrow maps 1 input object to 1 output object
• But transformation $T$ occurs in model synthesis:

$$T: O_1, O_2, O_3 \rightarrow O_4, O_5$$

• Ans: create tuple of objects, which is itself an object

$$O_{123} = \left[ O_1, O_2, O_3 \right]$$
$$O_{45} = \left[ O_4, O_5 \right]$$
Internal Diagrams are a category of points and arrows, also known as categories. A point is a domain with a single program.
• Design of an artifact is a meta-expression
  • synthesis is meta-expression evaluation
  • RQO paradigm

\[ b_5 = \text{javac} \cdot \text{M2TX} \cdot \text{M2MX} \cdot m_5 \]
• Categories lie at the heart of MDE
  • found at all levels in an MDE architecture
  • MDE is categories on an industrial scale

• Informally, categories provide a compact set of ideas to express relationships that arise among objects in MDE
  • language and terminology for MDE $D \times T$
  • can use CT more formally
    (e.g., Meseguer, Ehrig, Täntzer, Diskin, Czarnecki, …)

• Now let’s look for categories in Software Product Lines
#2: Categories in SPLs
SPL Overview

• SPL is a set of similar programs

• Programs are defined by **features**
  • increment in program functionality that customers use to distinguish one program from another

• Programs are related by features
  • program P is derived from program G by adding feature F
  • from our 1st lecture, a feature is a function:

\[ P = F(G) \]
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}

class gui {
    JButton format = new JButton(“format”);
    JButton add = new JButton(“+”);
    JButton sub = new JButton(“-”);

    void initGui() {
        ContentPane.add( format );
        ContentPane.add( add );
        ContentPane.add( sub );
    }
    void initListeners() {
        add.addActionListener(...);
        sub.addActionListener(...);
    }
    void formatResultString() {...}
}
Scale Reminder (From Lecture #1)

• 1986 database systems 75K LOC
• 1989 network protocols
• 1993 data structures
• 1994 avionics
• 1997 extensible Java precompilers 35K LOC
• 1998 radio ergonomics
• 2000 program verification tools
• 2001 verified compiler for Java1.0
• 2002 fire support simulators
• 2003 AHEAD tool suite 250K LOC
• 2004 robotics controllers
• 2006 web portlets
• 2008 SGI+JavaScript application
• 2009 ZipMe compression library
Perspective on Product Lines

• SPL is a **finite** set of similar programs

• Is **miniscule** subset of a domain

• Infinite set of SPLs in a domain
• SPL defines relationships between its programs
  • how are programs related?
  • by arrows, of course!
  • each arrow is a feature

• Empty program (0) may or may not be part of SPL
• Program design is a meta-expression
  • RQO paradigm
  • programs can have multiple designs

\[ p_3 = \text{format} \cdot \text{sub} \cdot \text{base} \]

\[ p_3 = \text{sub} \cdot \text{format} \cdot \text{base} \]

evaluating both meta-expressions yields the same program

format, sub are commutative
A Product Line is also a Category

- Category
  - point is a domain with a single program in it

- Has implied identity arrows

- Has implied composed arrows, as required
Fundamental Ideas in SPL Implementations
(devoid of implementation details)
Implementing SPLs

want this:

know this:

• Same function being applied to different inputs
Implementing SPLs

• Just store arrows once and reuse!
  • $n$ optional features, $2^n$ possible programs
  • compact representation of an SPL

store this:
Models of SPLs

• Implement a set of arrows
  • by transformations, superposition, or whatever
• Feature model defines legal compositions of arrows
• Yields a product line

![Diagram showing the process of combining models to form a product line](image)
Recursion

- SPLs can appear at any level of an MDE architecture
  - arrow adds same feature to an infinite domain of programs

- **Superposition** is a standard technique, but not always sufficient
  - Kästner’s CIDE (preprocessors)
Essential Distinctions of SPL and MDE

- An MDE “design” is a category with objects (domains) that are infinite in size
  - ex: metamodel of all class diagrams
  - # of such diagrams is infinite

- An SPL “design” is a category with objects (domains) that are finite in size
  - feature model defines a finite set of programs
Recap

• Categories lie at the heart of Software Product Lines
  • SPLs appear at all levels of an MDE architecture

• Informally, categories provides a compact set of ideas to express relationships that arise among objects in SPL
  • places in perspective what MDE and SPL communities have been doing
  • fundamental concepts that our tools need to support

• Next topic: model-driven product lines (MDPLs)
#3: Categories in MDPLs

Exposing fundamental verification and optimization relationships
Recall Commuting Diagrams

- Fundamental concept in category theory
  - all paths between two objects yield same result
  - theorems of CT

\[ f_1 \circ d_2 = d_1 \circ f_2 \]
Want to map a product line of S models to its corresponding product line of B models
- typical MDE transformations map only points, not arrows

**Operator** $\tau$ maps arrow $F_s$ to arrow $F_b$: $\tau(F_s) = F_b$
How Commuting Diagrams Arise

MSC

M2MX

s1 s3

s2

M2TX

Java

j1 j2 j3

javac

ByteCode

b1 b2 b3

can see commuting diagrams in this figure
Functors

- Are fundamental to Category Theory
- \( F: A \rightarrow B \) is an embedding of category A into category B
  - each object of A is mapped to an object in B
  - each arrow of A is mapped to an arrow of B such that arrow compositions in A are preserved in B
• I have encountered are isomorphic (A looks just like B)

• Product line of Java files → product line of bytecodes

• We have seen functors before in this tutorial
How Functors Arise

note: must map both objects and arrows to have a functor and commuting diagram

can see functors in this figure
Functors and Commuting Diagrams in PinkCreek

• Trujillo, et al. ICSE 2007

• Portlet synthesis

• Transform state chart into a series of lower level representations until Java and JSP code reached

- statechart of portlet
- this is a category!
- java code of portlet
- jsp code of portlet
Functors and Commuting Diagrams in PinkCreek

- When a feature is added, each representation is extended (arrows remain the same)

- Features are functors: map each object, arrow of the original category to those of an isomorphic category

- Feature composition = Functor composition
• Arrows are easy to draw…
  
  • may (or may not) be easy to implement
  
  • may (or may not) be practical to implement
  
  • **CT is not constructive – it doesn’t say how to implement arrows**
  
  • no more than UML class diagrams tell you how to implement a method
  
  • Tells you certain relationships exist, and if you can implement arrows, you can exploit commuting diagrams
More Examples that Exploit Functors and Commuting Diagrams
We found other uses for commuting diagrams and arrow operators in MDPLs:

- simplifying implementation (ICMT 2008, SOSYM 2010)
- improving test generation (SIGSOFT 2007, TSE 2010)
- understanding feature interactions (GPCE 2008)
- understanding AHEAD (GPCE 2008)

Briefly review the first two of these…
General Technique for Implementing MDPLs

appreciate use of categories to explain what is going on
Example 1: SOSYM 2010 Paper

- Work with G. Freeman and G. Lavender
- MDPL of applications written in SVG and JavaScript
  - to customize an application (removing, adding charts, controls)
• Created a set of arrows and a feature model for our MDPL
  • red arrows (defining a product line of charts) were tedious to write
  • created DSL for charts, where arrows were easy to write, compose
  • defined an operator $\tau$ to map 1:1 from green arrows to red arrows

Example 1: SOSYM 2010 Paper

```
feature model
SPL
```

DSL for chart arrows

```
\text{Cat- 50}
```

“lifted arrows”

$\tau$

operator $\tau$

feature model

SPL
τ Mapping of Arrow to Arrow

point-cut (AOP terminology)

GREEN Arrow

<xr:at select="/chart[@data-type='age-population' ..."
    <xr:append>
        <item attr="AGE_18_21" color="cyan" ...>
            </xr:append>
    </xr:at>
</xr:refine>

<xr:refine ...
    <xr:at select="/function[@data-type='age-population']
        @[parentId='ChartArea2'][@name='buildData']"...>
        <xr:append>
            <statement>
                this.chartAttrArray.push("AGE_18_21");
                this.chartNameArray.push("18-21");
                this.chartColorArray.push("cyan");
            </statement>
        </xr:append>
    </xr:at>
</xr:refine>

RED Arrow

advice

uses functional aspects
• Same result if we compose green arrows and translate OR we translate green arrows, and compose red arrows

• **Homomorphism** – mapping of expression in one algebra (GREEN) to a corresponding expression in another (RED)
Same result if we compose green arrows and translate OR we translate green arrows, and compose red arrows

Homomorphism – mapping of expression in one algebra (GREEN) to a corresponding expression in another (RED)

Diagram Constraints

\[ \tau(G_1 \cdot G_2) = \tau(G_1) \cdot \tau(G_2) \]
\[ = R_1 \cdot R_2 \]

Verification condition: our implementation is correct if this equality holds!
Initially our tools did not satisfy diagram constraints

- equalities of homomorphisms didn’t hold
- our tools had bugs – we had to fix our tools

now we have greater confidence in tools because they implement explicit relationships of domain models

win from engineering perspective

- insight into domains that we didn’t have before
- by imposing categorical structure on our domain, we have better understanding, better models, and better tools

Lifting is not specific to our application, it is a general technique for building MDPLs
Test Generation for MDPLs
• Work with E. Uzuncaova and S. Khurshid (ECE@UTexas)

• Testing SPLs is a basic problem
  • we can generate different programs, but how do we know that the programs are correct?

• Specification-based testing can be effective
  • start with a spec (model) of program
  • automatically derive tests
  • Alloy is example
Conventional Test Generation

Challenge: is there a $\tau$ operator?
Incremental Test Generation

- Spec
- Solutions

Conventional approach:
- S0 → A0
- S1 → A1
- S2 → A2
- S3 → A3

Incremental approach:
- S0 → A0 via τ
- S1 → A1 via τ
- S2 → A2 via τ
- S3 → A3 via τ

Testera → T3
Implementing $\tau$

- Spec S1
  \[ (A \lor B) \land (\neg A \lor C) \] // 20K clauses
  
a solution: $[A,B,C] = [1, 0, 1]$

- Spec S2
  \[ (A \lor B) \land (\neg A \lor C) \land (D \lor \neg A) \]
  
a solution: $[A,B,C,D] = [1, 0, 1, 1]$

- Solution for S1 “bounds” solution for S2
  - sound, complete

- Reason for efficiency…
Preliminary Results are Encouraging

• In product lines that we examined (typical of Alloy research), majority of cases incremental approach is faster

• 30-50× faster

• can now solve larger problems with Alloy

• See paper(s) for details
Recap

• An SPL or MDE application is an industrial-sized category

• Putting them together reveals foundational ideas of categories – commuting diagrams and functors
  • involves mapping both objects of a category AND arrows
  • need operators (transformation – to – transformation maps)

• Can exploit exposed relationships as
  • verification conditions
  • optimization possibilities
#4: Design Optimization

Frontier of D×T
Principles of D×T

- Design = meta-expression
- Synthesis = meta-expression evaluation
- Design Optimization = meta-expression optimization
  - find program that satisfies functional requirements and optimizes non-functional properties (performance, energy consumption)
I know of few examples of design optimization …

- relational query optimization (1980s)
- data structure optimizations (1990s)
- Neema’s work on synthesizing adaptive computing (2001)
- Püschel’s & van de Geign’s numerical library synthesis (2006)
- Benavides work on configurators (2005)
- …

Main challenge: finding domains where there are different ways to implement the same functionality

- commuting diagrams
- this is where design optimization occurs

If you think in terms of arrows, you have a conceptual framework and tools to explain and address design optimization in a principled, non-ad hoc way
Conclusions
Role of Mathematics in Design

• RQO helped bring database systems out of stone age

• Relational Model was based on set theory
  • this was the key to understanding a modern view of databases
  • set theory used was shallow
  • fortunate for programmers and database users
  • set select, union, join, intersect
  • disappointment for mathematicians

• $D\times T$ uses category theory
  • provides a language to express our results
  • places research results in context
  • new insight on verification, optimization issues
  • whether theorems from CT are applicable, I don’t know
Key To Success

- Educational benefit of the connection
  - common and simple language
  - offers new perspectives

- How often in MDE, SPL, MDPL do commuting diagrams arise?
  - don’t know; too early
  - but if you look, you’ll find them
  - theory says they exist
  - whether creation of operators practical depends on domain

Look for Them!
Lecture 3:
Extraction of MDE Architectures from Parallel Streaming Applications

work with Taylor Riché
Introduction

• Challenge of re-engineering complex streaming applications into a component-and-connector architecture
  • to eventually re-implement on a MDE platform

  Asynchronous Crash Fault Tolerant (ACFT) servers
  Classic Parallel Join of DBMS machines

• They were so complicated, we needed a way to convince ourselves and others that we understood their designs
  • we were not domain experts – not obvious how and why they worked

• We needed a structured way to explain and build our versions of these systems
• Classical & fundamental way to control design complexity
  • our work builds on results of a long line of pioneers
    (Labview 80, Gorlick 92, Broy 92, Moriconi 94, Garlan 96, Rumpe 97, Kong 03, Clarke 06, …)
    • use a standard component-connector model of application
    • elaborate it by simple transformations called **refinements**
Refinement is Not Enough!

- **Extend**: extend semantics of an application
  - adding new ports, components, connectors

- **Optimize**: break abstraction boundaries to achieve efficiency or availability
  - domain-specific optimizations to build modern streaming applications
• **Model Driven Engineering (MDE) Architecture**
  
  • start with executable model of a sequential component-connector architecture
  
  • transform it by *refining, optimizing, extending* to derive an executable parallel architecture that faithfully captures decisions made by domain experts
  
  • result is easy-to-understand *description* + *prescription* to recreate system on an MDE platform
Case Studies are known for their contributions to fault-tolerance and database machines
  - designs were never conceived in terms of transformations
  - novelty of their designs can be expressed by transformations
  - why certain transformations use is part of genius of their designers

No substitute for domain expertise
  - we use transformations to encode deep domain knowledge
  - express designs by transformations is novel to domain-experts, but the end result is rarely surprising
  - progressively revealing details is so straightforward that even non-domain experts (undergraduates) can follow along
  - although our descriptions are deceptively simple, it takes effort
  - ideal for teaching complex designs to others
Connection to Prior Lectures

• Incremental Design = SWD

• Express designs by compositions of transformations

**Design by Transformation**

• You’ll see elements of product lines, commuting diagrams, and MDE all integrated this lecture
Series of Mini-Presentations

Basics of Streaming Architecture Designs → Asynchronous Crash-Fault-Tolerant (Stateful) Servers

Hash Joins in Database Machines → Conclusions

we are here
#1: Basics of Streaming Architectures
• **Component-Connector Architecture** is a directed graph
  - box component or computation
  - connector communication path for messages, tuples
drawn in direction of data flow

• Semantics of box is clear from context

```
<table>
<thead>
<tr>
<th>1, 50, 2, 62, 53</th>
</tr>
</thead>
</table>
```

• Elide unnecessary details (sort key, sort order, sort type)

```
<table>
<thead>
<tr>
<th>1, 2, 50, 53, 62</th>
</tr>
</thead>
</table>
```

SWD - 10
Refinement

A → SORT → SORT(A)

map-reduce parallelizing refinement

A → HSPLIT → ... → OMERGE → SORT(A)

SORT

SORT

A

A_1

A_n

A'_1

A'_n
Transformations

• Refinement is a **transformation**
  • input pattern $\rightarrow$ output pattern

• All **transformations** or compositions of xforms in this talk have been proven correct
  • simple enough that intuition suffices
  • sometimes need Ph.D.

• **Correct by Construction**
Optimizations

- Break encapsulations to achieve non-functional properties
  - efficiency or availability

Same HSPLIT box (hash same attributes, same hash function)
Rotations

- Optimizations that reorder *stateless* computations
  - ex: property that each $A_i$ message is assigned to a single $B_j$ stream

$m$ streams are merged

$k$ streams are produced
Box Extensions

• Extend the capabilities (semantics) of a box

• Extensions add “features”

• Accomplished by preprocessors #ifdef inclusion of extra code

• Or by more sophisticated means
• Example: Webserver takes sorted tuples and creates a web page of sorted results

• Extend Webserver & Sort with new ports and add a feedback data flow called newkey which changes the key that sort uses
  • switching from artist names to album titles
  • switching from last names to SS#
Always Executable

- User supplies boxes and tests for boxes
- Can reuse tests for boxes after refinement
  - logically cannot distinguish input-output response of a single SORT box from its parallel counterpart
- Similar arguments for extensions and optimizations
Recap

• Component-connector architecture is implementation model
  • transformations progressively elaborate models by refinement, extension, or optimization
  • result is always executable

• Now, let look at some examples…
#2: Asynchronous Crash-Fault-Tolerant (Stateful) Servers
Overview

- Sequential server architecture has a cylinder topology
- Unroll cylinder by breaking along the seam
Our Goal

- Transform a (stateful) server that works in the ideal world of synchronous networks and no box failures to an

**Asynchronous, Crash-Fault-Tolerant (stateful) Server**

- consider Synchronous CFT transformations first
- Asynchronous (recovery) transformations last
Basics on Crash-Fault-Tolerance

• Ability of a server to survive a number of failures

• **Failure** – when a box stops processing messages
  • no messages pass through a failed box
  • a failed box cannot create new messages
  • *assume each box executes on its own machine*
    – multiple boxes can run on single machine
    – if machine fails, all boxes on that machine fail
    – failures do not propagate across machine boundaries
Standard Failure Assumptions

• Failures of network components
  
  serializer (▲)
  demultiplexor (▼)
  reliable broadcast (●)

affect a machine the same as pure software boxes
  • ex: a machine can’t process requests if its network card stops working

• Ultimately we not depend on synchronous networks
  • do expect eventual synchrony
  • use **retransmissions** (in application, network protocol, or both) to deal with transient packet loss

• Requests are benign; BFT removes these assumptions
Technical Goal of CFT

- Eliminate **Single Points of Failure (SPoF)**
  - a failure of a single box (machine) causing the server abstraction to fail
  - our current design has 3 SPoFs

- “Solve” problem by replicating boxes
  - not only solution – we follow most advanced solution to date
  - appeared in SOSP 2009
Step 1: Agreement

- Add an agreement node $A^\perp$
- $A^\perp$ materializes implicit network message queue, passing messages one at a time to the server
- In effect, $A^\perp$ does nothing it is a place holder for later refinements

Next steps replicate $S, A^\perp$ boxes
Step 2a: Replicate Servers

- Make $k$ copies of server
- Each server receives exactly the same sequence of messages from the $A^\perp$ abstraction
- $QS$ collects a quorum of identical messages; transmits message when a sufficient number of copies are received
- *Refinement emulates abstraction of a single correct server*
Why are $k$ Server Replicas Needed?

- To tolerate failures of server boxes
  
  tolerate $f$ failures, $k = f + 1$

Step 2b: Replicate $A^\perp$ Boxes

- Make $m$ copies of $A^\perp$
- Client requests are routed by box Rt to *some or all* $A$ replicas
- $A$ replicas run an **agreement protocol** (Lamport 1998) to decide which is the next client message to process
- $A$ replicas vote and a quorum is taken by QA; when a sufficient number of identical messages is received, QA forwards a single message
- *Refinement emulates abstraction of a single queue*
Why $m$ Replicas of A are Needed?

1. Ans#1: tolerate failures of A boxes

   tolerate $f$ failures, need $m = 2f+1$

2. Ans#2: a consistent order of requests is essential for correct server behavior

   If $S$ replicas processed client requests in different orders, server replica states would diverge and responses from different servers for a single client request would be inconsistent

   Inconsistencies violate the one-correct-server abstraction
Where We Are...
Where We Are Going

- Dissolve existing module boundaries
- Define new (green) abstractions
- Apply rotations to eliminate SPoFs
Step 3a: (►, Rt) Rotation

- Each client request is sent to a subset of A replicas
Step 3b: (▶, Q, ⋆) Rotation

- Each quorum-decided request from replicated A boxes is delivered to all Server replicas
Step 3c: (▷, QS, ◁) Rotation

- Each quorum-decided response from replicated S boxes is received by a client box.
Why So Simple?

- All rotations involve stateless computations
  - state would require a heavy-weight solution (agreements, etc.)
One More Optimization

- Reliable broadcast is very expensive
- Under the right conditions (e.g., quorums) reliable broadcast (●) can be replaced with unreliable broadcast (☉) which is easy to implement

![Diagram showing the comparison between reliable and unreliable broadcast systems.]
Final Synchronized CFT Result

No SPoF

SCFT
Really Quick Tour of Recovery (Async) Transformations
Overview

• Just as databases can recover from machine failures, so too can servers

• Recovery limits the situations where clients see unresponsive server abstraction
Where We Are…

SCFT

ACFT

SWD - 40
Extension of SCFT to ACFT

Servers now talk to A boxes (new connectors are added)
Extension of SCFT to ACFT

SCFT Server Architecture

A and S boxes are extended (new ports, capabilities added)

ACFT Server Architecture

incrementally adding features to existing boxes
• Rotations were unfamiliar to our domain experts

• Their informal designs jumped directly from

to the one using a reliable crossbar which we derived

SWD - 43
Quick Recap

• Incrementally recovered a design created by experts to map a vanilla server to an asynchronous (recoverable), crash-fault-tolerant server
  • starting from a simple client-server model and progressively transforming it into the target architecture

• **LOTS of engineering left**
  • but now we have the architectural plans to recreate it incrementally and in a way that is easy to understand

• Now look at a very different domain where a sequential architecture is mapped to a parallel architecture using **exactly the same principles**
#3: Parallelizing Hash Joins in Database Machines
Gamma Database Machine

• Gamma was (maybe still is) the most sophisticated relational database machine ever built in academics
  • University of Wisconsin late 1980’s early 1990s

• Look at how hash joins were parallelized
  • fundamental result in parallelizing joins
  • representative of commercial systems today
  • presented in a new way
  • derive Gamma hash join architecture from first principles
Hash join takes 2 streams of tuples (A,B) as input and produces the join of these streams (A*B).

Algorithm:
- read all of stream A into memory in a hash table
- read B stream one record at a time; hash B’s record and join it to all A record’s with the same key
- linear algorithm

How did Gamma’s Designers parallelize HJOIN?
Next Slides Explain Derivation

\[ A \xrightarrow{HJOIN} A^*B \]

\[ B \xrightarrow{HJOIN} A^*B \]
First Refinement

- Because joins are the most complex operator, increase efficiency by reducing the size of its input streams.

- Used Bloom filters to eliminate B tuples that do not join with A tuples.
• Bloom filtering is a common technique for disqualifying tuples from further processing

• Algorithm:
  • clear bit map $M$
  • read each $A$ tuple, hash its key, and mark corresponding bit in $M$
  • output each tuple $A$
  • after all $A$ tuples read, output $M$
• The filtering part of Bloom filters
  • eliminates B tuples that cannot join with A tuples

• Algorithm:
  • read bit map M
  • read each tuple of B, hash its key: if corresponding bit in M is not set discard tuple (as it will never join with A tuples)
  • else output tuple
The First Refinement

- Expose inner details of HJOIN box
- Can prove correctness of this refinement
Parallelize Each Box in this Architecture:
Parallelization of BLOOM Box

- Algorithm:
  - HSPLIT stream A
  - compute Bloom filter on each substream
  - reconstitute stream A
  - form merge bit maps to produce single bit map M
## Parallelization of BFILTER Box

- **Algorithm:**
  - split $M$ into $M_1 \ldots M_n$ and distribute
  - hash split stream $B$
  - filter $B$ substreams in parallel
  - reconstitute stream $B'$

always hash split streams $A$ and $B$ using the same hash function! This gives us properties on which to optimize!
Parallelization of HJOIN Box

- Algorithm:
  - split both streams using same hash function
  - A and B tuples can join only if they have the same hash key
  - perform $n$ joins (rather than $n^2$) in parallel
  - reconstitute join
• Substitute parallel implementations for each box
• Note 3 optimizations are possible
• Here are the first two…
Stream A is hash split into $A_1 \ldots A_n$, reconstituted, then hash split again.
MERGE – HSPLIT combination is the identity map.
Optimization – get rid of MERGE-HSPLIT

Same for stream B.
Applying Optimizing Rewrite

HJOIN

- Still one more optimization to perform…
The (Almost) Final Design

- Elegant
- Easier to remember the derivation than the design itself (!)
- Each step can be proven correct, so the final design is correct
- Not whole picture: rotations rewrites are applied when HJOIN boxes are composed
  see our paper or original Gamma papers
#4: Conclusions and Future Work
Recap

• Showed how SWD used to extract an MDE architecture from parallel streaming architectures
  • architectures are always executable, derived top-down
  • *architecture is a D\times T* meta-expression

• Used traditional technique of refinement + box and model extensions and optimizations – all are needed to explain the complexities of modern streaming architectures

• Encoded deep domain knowledge by simple xforms, demonstrated approach with 2 case studies, and validated our approach by manually re-creating them in the incremental manner in which we presented them
Although our MDE architectures look simple
  • it took effort to refresh our domain knowledge
  • polish core abstractions and transformations

Worth the effort:
  • complex designs can be explained in a simple way
  • can be appreciated by non-experts
  • techniques (and these examples) can be taught to undergraduates

Incremental Design is not just “cute”
  • ultimately indispensable for future software development technologies that eventually integrate design, construction, verification, and testing

\(D \times T\) is at the heart of all of this
Thank You!