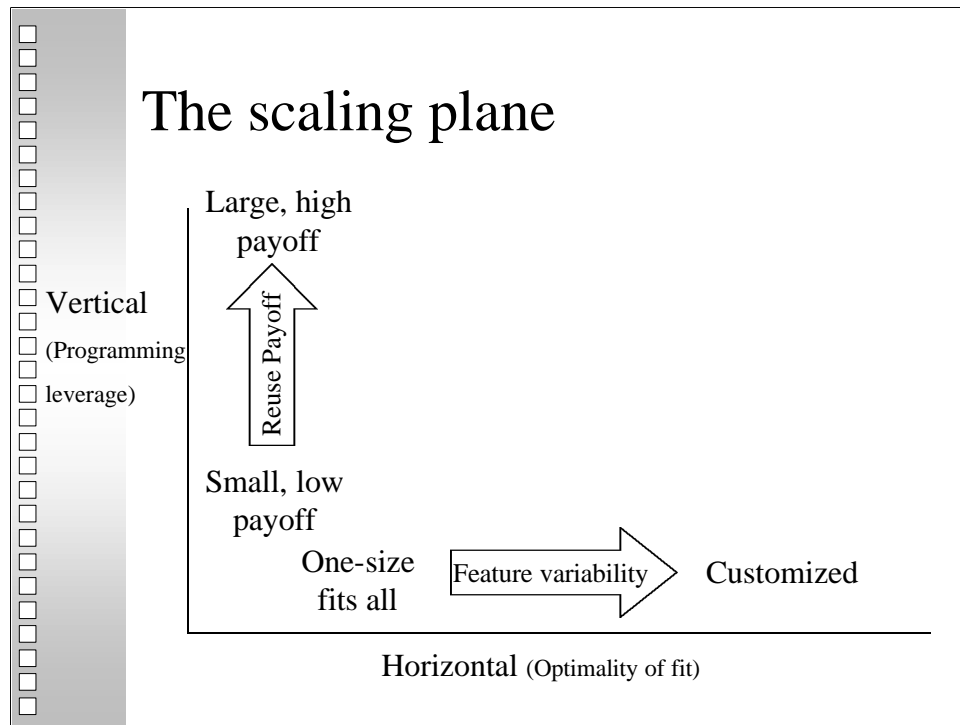


I want to characterize various categories of reuse technologies in terms of their underlying architectures, kinds of problems that they handle well, and the kinds of problems that they do not handle well. In the end, I want to express these operational envelopes as descriptions of their niches. Because of the significant architectural differences in the various technologies, it is difficult to find a common framework from which to compare them. However, there are several operational parameters on which they can be compared and that distinguish the key operational characteristics that are of most importance to the user of the technology.




As an organizing framework for the niches, I will characterize them along two important dimensions of scaling: 1) how well they scale up in terms of raw size and thereby **programming leverage**, which I will call **vertical scaling**, and 2) how well they scale up in terms of **feature variation**, which I will call **horizontal scaling**. These two dimensions are typically fundamentally opposed to each other.

In some sense, vertical scaling is a productivity measure that defines how many units of functionality one gets out per unit of effort. Big reusable parts increase this measure of efficiency and therefore, this dimension correlates strongly with component size. While there are other parameters (i.e., the percentage of domain coverage with respect to an application), component size is a serviceable if approximate proxy for vertical scaling.

On the other hand, as parts increase in size they incorporate more and more design decisions, which we can view as limits on the contexts in which they fit, on the contexts in which they meet all of the requirements. Thus, the horizontal dimension is a measure of the reusability of the components across a range of different applications. The more customized a component is to a narrowly defined set of requirements, the less likely it is to be reused and therefore, the less likely there will be enough reuses of it to recoup the cost of building it.

The notional plane defined by these two dimensions is the scaling plane and serves to emphasize the key differentiating features and characteristics of various technological approaches to reuse. Of course, there are other important dimensions in this notional space, notably the performance of the target code. And we will consider performance as a factor that serves to **constrain the niche footprint** in the scaling plane.



# Reuse Technologies

- Concrete components
  - ✦ e.g., Functions, OOP, Frameworks, DCOM, ...

Concrete components are those that 1) are written in **conventional programming languages**, 2) are **internally immutable**, and 3) represent a **one-size-fits-all style** of reuse. They include such categories as functions, Object Oriented classes, frameworks, and COM-like middleware components. They often exhibit serious reuse flaws such as inadequate performance, missing functionality, inadequately populated libraries, there are design incompatibilities, etc.

Except for a few niches, concrete components have not lived up to their hype. I believe that this is a **fundamental flaw of concrete components** that arises from the programming languages in which they are written. Conventional programming languages (e.g., C, C++, Java, etc.) all require too many low level, detailed decisions to be made too early. And thus, concrete components typically do not fit their requirements very well.

In particular, ....

## Concrete Components

- Small components don't work well
  - ◆ Assembly work per unit payoff is low
- Small components specialized niches OK
  - ◆ Overhead masked by human response (e.g., UI)
  - ◆ Standards or domain narrowness mitigate
- Big components work well
  - ◆ Programs or subsystems
  - ◆ High programming leverage
  - ◆ Payoff dominates overhead costs

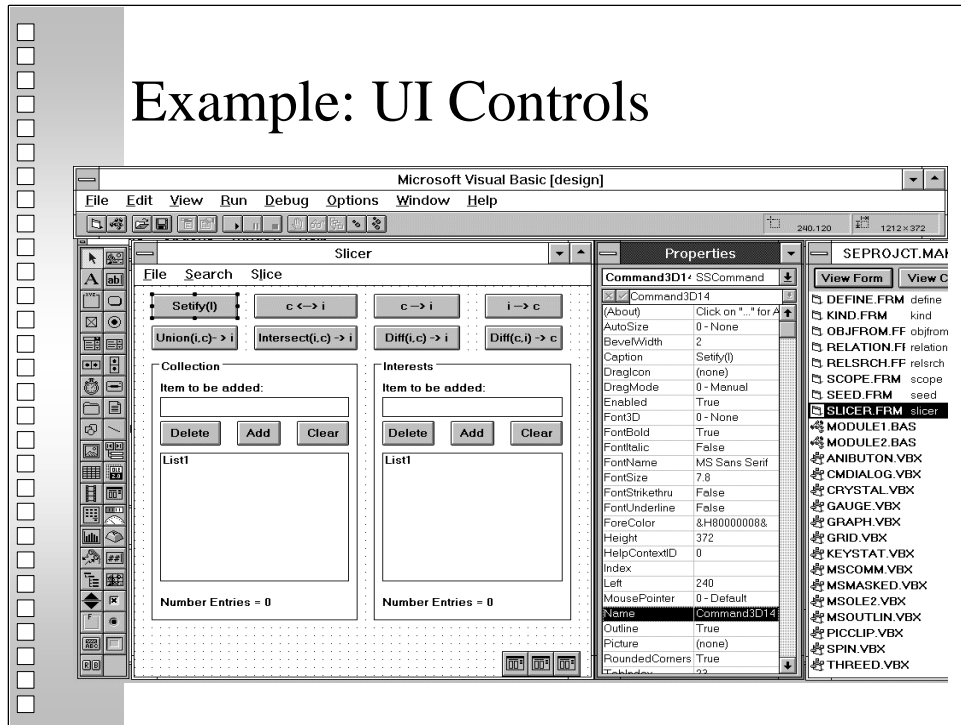
**Small, broad-spectrum:** Libraries of random small, broad-spectrum components **work poorly** well because 1) the effort involved in assembling them into applications mitigates the profit gained by reusing them and 2) broad-spectrum components don't really address the specialized requirements of most application areas. This does not mean that you should not use libraries. It means that you should not expect them to cover 80 or 90% of your application. If you get a 10% coverage, you are lucky.

**Small components with mitigating circumstances:** The second niche is smaller-scale components (e.g., as UI components) that can achieve high customization via compositionally induced variation and yet still exhibit adequate performance in spite of the compositionally induced overhead. Often this is because the overhead is masked by other factors such as the human response times. This niche **trades performance degradation** (which may be masked) **for high levels of adaptability**. In addition, there are often direct manipulation tools that reduces the effort of assembling the components thereby boosting the reuse payoff. The UI is a good example of this class. **SEE NEXT SLIDE.**

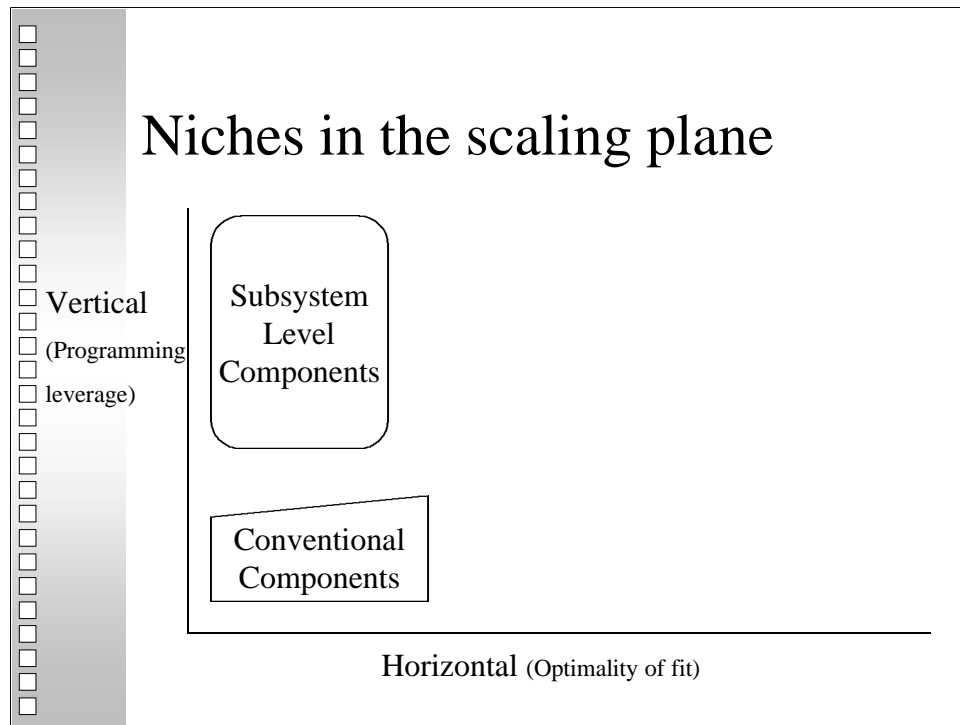
**Specialized niche:** The third niche is where **standards or domain specific requirements** have been so **narrowly defined** that one-size-fits-all components are satisfactory. Their weakness is shelf life since their reusability declines as the standards are undermined by change. Again, UI components are a good example of this a niche determined by narrow standards and a library of actuarial and statistics routines would be an example of a domain specific niche.

**Very-large scale:** However, concrete components do succeed well in a few niches. The first niche is very large-scale components that **just happen to fit the programmer's** needs or are designed to a standard that predestines a good fit. These trade customized fit and broad reusability for high programming leverage. The downside of very large components is that it is just too expensive to build libraries of them that cover all circumstances. The requirements are spread too broadly across the field of applications.

# Example: UI Controls



Tools can make the assembly process low cost and therefore boost the profit due to reuse. So, you get a reasonable profit from reuse, good horizontal scaling through the variability allowed by composition, a tool that makes composition of components cheap, and a context of usage where performance degradation induced by the generality of the componentry is masked by other factors, such as slow human response times.



So, in conclusion, there are a couple of islands where concrete components work well or, at least, acceptably: Program or subsystem level components, and specialized islands where tools or circumstances allow a reasonable combination of vertical and horizontal scaling without onerous costs. But these are rare.

**Technical basis –**

Piece part representation = programming language constructs

Assembly mechanism = programming language control structures

So, what can be done about these downsides???



## Essence of The Technology

Class

Elements

Operations

Concrete Reuse

PL Struct.

Hand Assem.



## Reuse Technologies

### ■ Concrete components

✦ e.g., Functions, OOP, Frameworks, DCOM, ...

### ■ Compositionally Derived Components

✦ e.g., Templates, GenVoca, ...

A fundamental difficulty of concrete components is the tension between optimality of component fit (horizontal scaling) and the need to scale the components up in size (vertical scaling) to achieve higher levels of programming productivity. People have begun to ask the question as to whether there is some way to factor the reusable library along lines orthogonal to conventional programming component lines (that is, orthogonal to subroutines, classes, methods, objects, etc.) such that the factors represent **aspects** of the design that can be assembled in different ways to get greater horizontal scalability while still retaining reasonable levels of vertical scaling. That is, one can **use big components** but **generate variations** on the general design to better fit the context in which they will be used. This would eliminate problems like poor performance, missing functionality, inadequately populated libraries, incompatible data organizations.

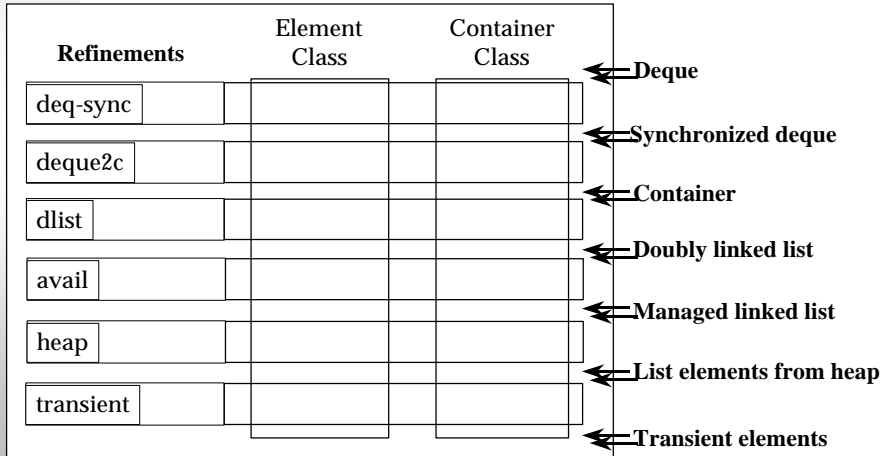
I am calling this class of work **compositionally derived components**. Templates and abstract classes are a weak form of this notion but they are overly restrictive and make accomplishing this kind of factorization extremely painful. There is work along these lines going on in the object oriented community and the generator community. Don Batory's GenVoca work (in the generator community) is a good example of this class of "reuse based generation system." There are very similar ideas being worked on in other research communities such as Notkin and VanHilst in SE and a number of people in the OO community.



# GenVoca derives a container

Type Equation

```
deque_usm = deq_sync [ deque2c [ dlist [ avail [ heap [ transient ]]]];
```



So, let's look at an example. We want to generate a two or three related of classes and all of their methods, which will provide a degree of vertical scaling. In addition, we want to be able to horizontally scale these parts by varying elements of their internal design such as, 1) whether or not they are represent concurrently shared data structures that must have some synchronization machinery, 2) the actual organization of the data structure, whether the data structure is persistent or transient, and so forth.

In Batory's GenVoca approach, these **factors** are expresses as layers in a **layers of abstraction model**. For any specific instance of the classes, a **type** equation describes the order of the layers. That is each layer encapsulates one and only one abstraction or feature of the target component. The top layer encapsulates synchronized deque-ness but defers making any design commitments with respect to other features such as boundedness or exact container characteristics. Each lower layer will introduce a new design decision that focuses on a (ideally) singular design decision. The layers are composed (or stacked) by writing a type equation that determines each design decision (i.e., refines the Realm by picking a Component implementation for that Realm). This is a top-down refinement strategy. It assembles the code for classes and methods by weaving together slices of the code drawn from the specific Components in the stack. Each component (e.g., dlist) is a step-wise refinement that maps a more abstract type into a more concrete type (e.g. container -> doubly linked list) adding detail as the refinement proceeds.

This holds some promise as a way to get around some of the problems of concrete component reuse, that is to say, to be able to get some degree of vertical scaling while improving the degree of horizontal scaling.

So where does this work well and where does it not work so well?

# Compositional Derivation

## ■ Example domain: Data structures

### ■ Benefits:

- ◆ Improve reuse by raising level of abstraction
- ◆ Combinatorial amplification of library (horz.)
- ◆ Generation is fast

### ■ Shortcomings:

- ◆ Level of abstraction still low
- ◆ Large grain components mitigate horz. scaling
- ◆ Dependencies limit extensibility
- ◆ Inter-component optimization difficult

**Good points:** Programming leverage through raising level of abstraction.

Combinatorial amplification of the reuse library mitigating the costs associated with trying to cover the small ground (horizontal scaling) with concrete components. Thus, increases the payoff of reuse.

The generator is relatively fast (compiler like performance).

**Shortcomings:** Raises the level of abstraction (and therefore vertical scaling) but not very far. We are now programming to more abstract data structures but not as abstract as we would really like.

#### **Technical basis –**

Piece part representation = abstracted programming language constructs – not all that abstract and relatively restrictive still

Assembly mechanism = inlining of programming language control structures

Optimization or reorganization mechanism = only via choosing alternative assembly of piece parts

And there is no reorganization or reweaving strategy

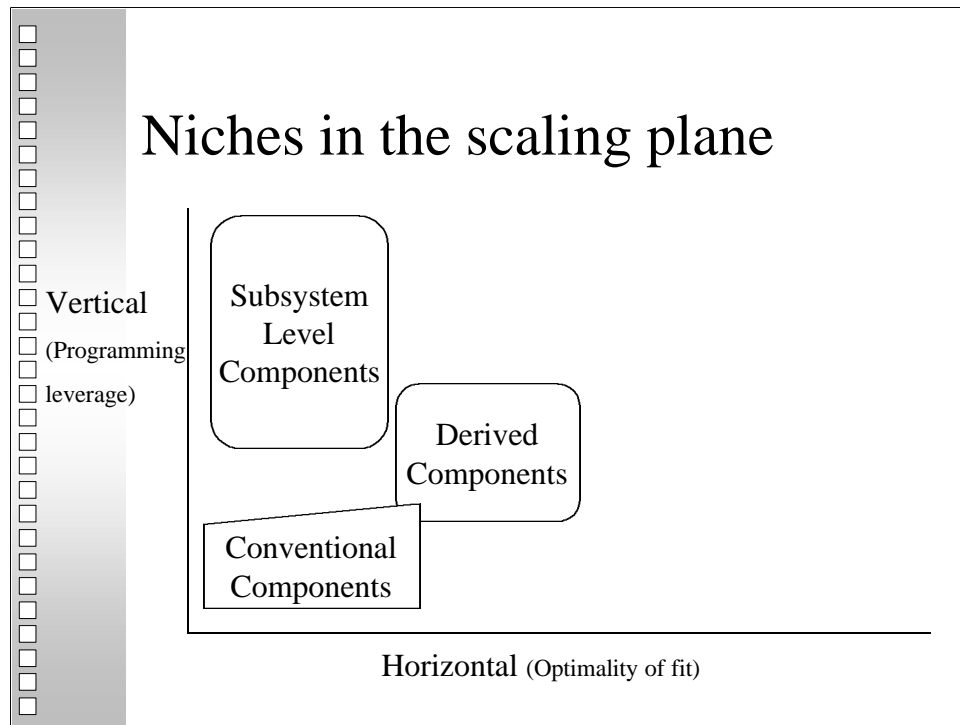
Does not have a good mechanism for dealing with global, inter-component the dependencies. Design logic is isolated by the components and that may cause inefficiencies. Code integration and sharing among distant layers not possible (or at least, very hard) -- that is abstraction layers survive in the application code and optimizations that might result from clever merging and reorganization of the low level code is not possible.

Factoring does not always completely separate components making them independent. Dependencies between supposedly independent components may be implicitly encoded with the components. In some sense, this is using engineering to handle hard dependency problems. For example, an upper level component may need to know the programming language type and size for a storage structure that will be local to a block of code that it will generate. But a lower level component may be responsible to determining that information. An ad hoc inter-component communication protocol has to be engineered to send that information from one component to the other. Conventional programming constructs like ~~parameters and returned values do not fit the model well.~~



## Essence of The Technology

<u>Class</u>		<u>Elements</u>	<u>Operations</u>
Concrete Reuse		PL Struct.	Hand Assem.
Composition		Abstract PL	Inlining



So, compositional derivation systems provide some improvement in our ability to simultaneously scale vertically and horizontally.

However, if the horizontal variability of the software is not easy to isolate within a singular layer (or equivalently, horizontal variability depends strongly on intra-layer dependencies) then the derived component model does not work well for such cases. For example, a search performance requirement in a high layer may have design effects in various layers. E.g., fast search of large containers may preclude implementation of those layers as a doubly linked list even tho' other requirements (e.g., fast insertion and deletion) may suggest a doubly linked list implementation. Similarly, machine architectural requirements may have global designs effects that require coordination of design variations within multiple layers. For example, stringent search performance requirements might require restructuring the design so that the search can be segmented and performed by multiple CPUs. Such design variations introduce inter-layer dependencies that require coordinated changes within several layers. The compositionally derived component strategy does not lend itself to this kind of horizontal scaling.



## Reuse Technologies

### ■ Concrete components

✦ e.g., Functions, OOP, Frameworks, DCOM, ...

### ■ Compositionally Derived Components

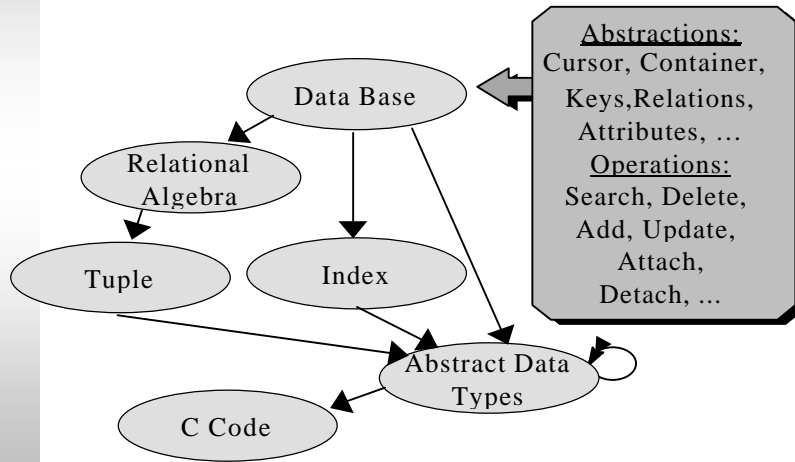
✦ e.g., Templates, GenVoca, ...

### ■ Pattern-directed transformation systems

✦ e.g., Draco, CAPE, IP, DMS, ...

Pattern-directed transformation-based generators allow greater degrees of customization (horizontal scaling) than composition-based generators because they use language-based building blocks that are less rigidly constrained than the components of composition generators. The underlying technology is usually a transformation engine that allows the developers to define (reusable) rewrite rules that transform a program written in a domain specific language into a conventional programming language like C. There are several example systems but I will focus on Jim Neighbors Draco as an exemplar of the class, not because I want to shill for Jim, but because it is a good exemplar, it is mature, has been applied to many domains, and has been used to build a commercial product (CAPE).

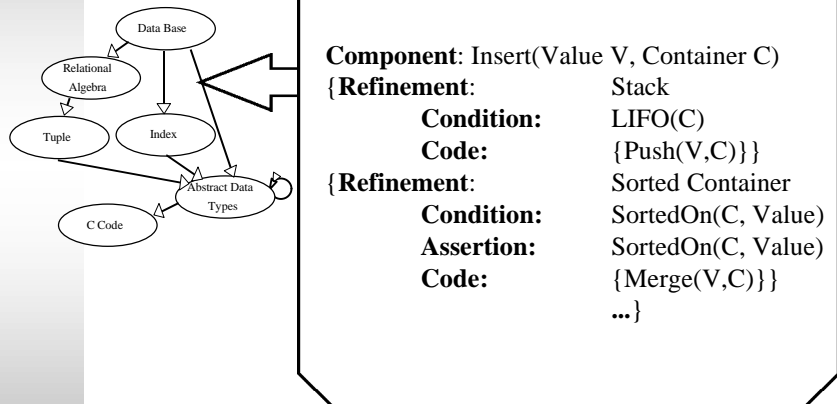
# Draco Domains



The basic Draco paradigm divides the world into domains each of which has its own mini-language (e.g., the relational algebra) in which programs can be written and components can be defined. In this somewhat contrived example, the domains are the data base domain, the relational algebra domain, etc. Within each domain, programs are written in abstractions and operations specific to that domain. For example, the data base domain has abstractions of Cursor, container etc. and operators of search, delete, etc.

The mini-languages are prescriptive (i.e., operational) rather than declarative. The generation paradigm is based on rules that map from program parts written in one or more mini-domain language into lower level mini-languages recursively until the whole program has been translated into the lowest level mini-domain of some conventional programming language (e.g., C, C++, or Java). Between translation stages, optimizations may be applied that reorganize the program for improved performance. These optimization steps are highly effective within a single domain because domain knowledge can be used to effect the optimization but they are far more difficult between domains.

# Refinements



DRACO refinements map the notation of one domain into the notation of one or more conceptually lower level domains. This is a top-down strategy for deriving the details of the code. In this example, the data base domain is mapped into the relational algebra, index, ADT domains. This is probably a bit of a hokey example in which the domains are overly inclusive, complex, and not particularly well thought out but it illustrates a variety of contrasting refinements, pre-refinement enabling conditions and post-refinement assertions. In this case, the data base instance might refine to a simple stack, a simple sorted container, or an indexed relation depending on the conditions.

Please note that refinement mappings are not strictly hierarchical. In fact, domains are frequently mutually recursive, which is the characteristic that makes DRACO like generational behaviors impossible to accomplish using the constructs of conventional programming languages (e.g., templates). Although refinements could probably be simulated with object oriented constructs (e.g., subclassing), without domain specific optimizing transforms, the performance would most certainly be abysmal and the parameter list “plumbing” of the methods across diverse refinements would at best be excessively complex and at worst be rococo.

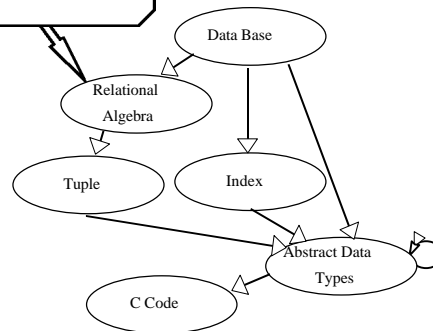
Also, note that for reasons of compact presentation, I have taken some liberties with the DRACO notational forms.

Now, cascades of refinements compiled directly and naively would likely generate very inefficient code, so DRACO uses transformation to eliminate the inefficiencies introduced by naïve generation (i.e., introduced because of the relative isolation of the individual refinements in the cascade).

# Optimizing Transforms

## Domain to self:

**Join(?Relation1, Empty\_Relation, ?Attribute)**  
=>Empty\_Relation  
**Select(?Relation1,TRUE\_expression)**  
=>Relation1  
**Select(?Relation1,FALSE\_expression)**  
=> Empty\_Relation



Transformations perform optimizations on domain notations, mapping a domain notation into the same domain notation. These transformations are used to clean up the inefficiencies introduced into generated code because of naïve code generation. Sweeping domain specific optimizations are often possible, optimizations that would be impossible at the code level because the higher level abstractions are no longer present at the code level. Trying such optimizations at the code level would require the compiler to infer the higher level abstractions from code, a generally impractical task. However, since DRACO's transforms have the domain specific abstractions in hand, they can therefore make transformations that often eliminate large chunks of code.

Often transforms are optimizing inefficient machine generated code, the kind of code that a person would never write but that program generators write all of the time. In the past, such inefficient code (generated because the left hand does not know what the right hand is doing) doomed generator code to poor performance. Domain specific transformations can eliminate this deficiency and generate code that is as good as or nearly as good as hand tailored code. The cost is slightly increased generation execution time.

This slide shows an example of optimizing transformations in the relational algebra domain. The first example expresses the idea that any relational expression (Relation1) joined with the empty relation can be replaced by the empty relation. Thus no join code need be included in the target program in this case.



## Pattern-Directed Transformations

- Example domain: communications protocols
- Benefits:
  - ◆ Fine grain rules allow great horizontal scaling
  - ◆ Mapping between language objects
  - ◆ Powerful in-domain optimizations
- Shortcomings:
  - ◆ Cross domain optimizations (interweavings) can explode search space

These techniques achieve significantly greater degrees of custom component fit for the target application (i.e., horizontal scaling) while simultaneously allowing scaling the size of the components. However, the cost is reduced target program performance because while the rules that reorganize/optimize the program at each stage (intra-domain optimizations) can, in theory, find the optimal reorganization, the search space for inter-domain optimizations is very large. So in practice, target program performance is often compromised. Nevertheless, there are many application domains for which the performance degradation is not onerous or may be an acceptable tradeoff for the vastly increased programming leverage. The CAPE system for generating communications protocols, which is based on DRACO [4], is an example of a domain where the tradeoff is acceptable.

### **Technical basis –**

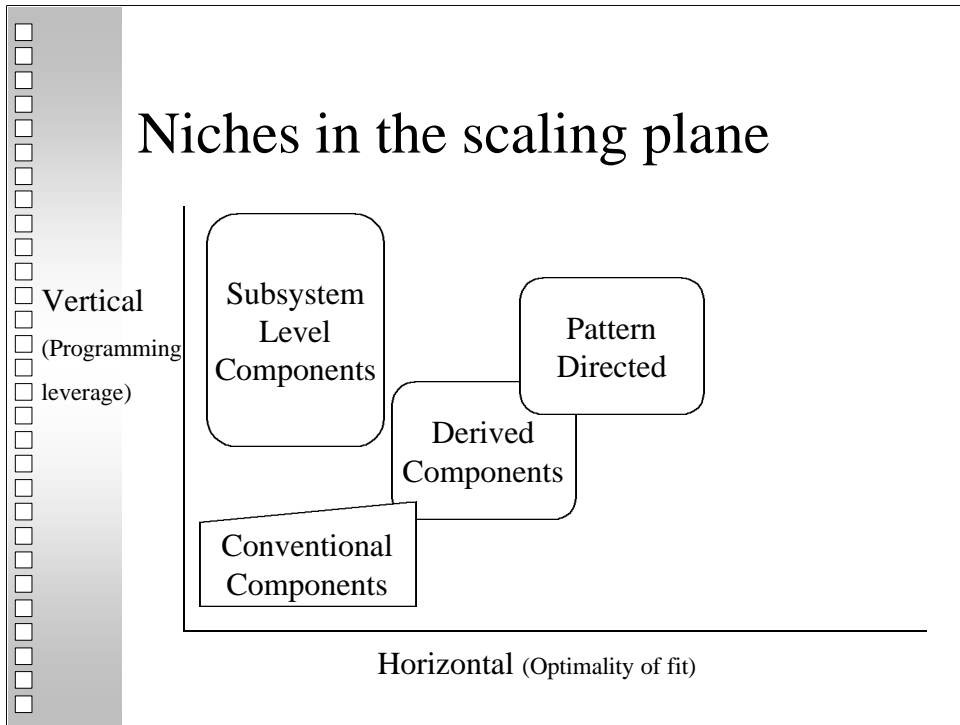
Piece part representation = domain language-based entities

Assembly mechanism = translation and merging of language-based piece parts via pattern-based mechanisms

Optimization or reorganization mechanism = only via pattern-directed mechanisms, which is a grossly inefficient way to effect reorganizations and accommodate remote dependencies (explodes search space)

## Essence of The Technology

<u>Class</u>		<u>Elements</u>	<u>Operations</u>
Concrete Reuse		PL Struct.	Hand Assem.
Composition		Abstract PL	Inlining
PD Generator		DSL Struct.	PD Xforms



So, pattern directed generator systems extend the scaling significantly in both directions.

But there is still this problem of how to get inter-component optimizations without exploding the search space so that the generators complete their work before the sun becomes a cold dark ember.



## Reuse Technologies

### ■ Concrete components

✦ e.g., Functions, OOP, Frameworks, DCOM, ...

### ■ Compositionally Derived Components

✦ e.g., Templates, GenVoca, ...

### ■ Pattern-directed transformation systems

✦ e.g., Draco, CAPE, IP, DMS, ...

### ■ Reorganizing generator systems

◆ AO generator, AOP, ...

And now we are in the realm of speculation about the future.

So, I believe that we need some additional generation machinery. Over the last year or so, I have been building a generator system to test the feasibility of doing inter-component rewavings without exploding the search space. It is called the Anticipatory Optimization Generator because it allows the user to abstractly anticipate interweavings and tell the generator about it. There is another similar effort called Aspect Oriented Programming although the underlying mechanisms are quite different.

## Problem: Antagonistic Goals

- High level operators and operands provide programming leverage & variations
  - ◆ E.g., (image  $\oplus$  neighborhood) convolution
- But fracture and de-localize code pieces
  - ◆  $b = ((a \oplus s)^2 + (a \oplus s')^2)^{1/2}$
  - ◆ Needed optimizations: code sharing, re-org. & re-weaving
- Conventional Optimization approaches induce large search spaces

The central difficulty revolves around delocalization of information. [Letovsky and Soloway] If I factor the operators and operands into highly general constructs, I can write combinatorially many compact expressions with them that effectively form an infinite *virtual library* of reusable components, one component for each possible composite expression. For example, the convolution operator  $\oplus$  defines the general structure of the convolution computation, i.e., it is a sum of pixels within a neighborhood times weights. The characteristics of the neighborhood is not defined by the convolution operator. They are defined by the template,  $s$  and  $s'$ , in the equation. That is, what are the dimensions of the neighborhood, how are the weights calculated and what variables do they depend on, and what if any special cases are there (e.g., are boundary cases processed differently from non-boundary cases). In this example, they are.  $S$  and  $sp$  define all of these SPECIFICS of the convolution operation.

However, this means that the tightly integrated information needed by the compiler to generate high performance code is split across many operators and operands. With current technology, compiling such expressions requires huge search spaces of possible transformation sequences to assure finding the optimal localizations for high performance execution.

On the other hand, if I define less general operators and operands in which cross operator code is already localized for performance reasons, the number of possible variations that can be produced by my generator drops precipitously and my infinite virtual library very likely becomes a finite library.

In summary, the problem is trying to achieve three goals simultaneously; 1) factoring domain into high leverage operators and operands, 2) compiling expressions of these operators and operands into high performance code, and 3) doing the compiling without engendering a large search space that renders the compilation algorithms impractical.

# AOG Features

- Schema language foundation
- Pattern-driven & tag-driven transforms
  - ◆ *Organized by inheritance type and by stage*
  - ◆ *Tags capture non-pattern-driven knowledge*
  - ◆ *Tags act like AST-localized interrupts for performing optimizations*
- TD's are large-grain programmatic transforms
- Specialists: Partial evaluator & inference engine
- No explicit loops
- Performance with small search spaces

AOG is built in terms of highly purposeful, large-grain programmatic transformations, e.g., transforms that create, place, and fuse loops. They operate on conceptual or logical patterns in an AST and achieve a level of insulation from the physical details and variations in the AST via a "schema" expressed in a Prolog-like language. These schemas perform pattern matching, parameter binding, enabling condition checking, and logical inference for the transforms. The schemas are first class items which are created dynamically, stored within the AOG's data structures, and executed by the transformations. The schema language has the major Prolog operators as well as extensions required for the generation task. It is fully backtracking thereby allows a schema execution to fail and then backup to the last choice point to try the next choice. This allows schema executions to explore all possible bindings in search of the preferred solution.

AOG does use conventional pattern-directed transformations for the initial stages of translation but they are organized into an inheritance hierarchy based on operator and operand abstractions. They are also organized into translation stages each of which seeks a narrowly defined translation purpose. (e.g., creating, placing and fusing loops).


For optimization stages, AOG uses tags on the AST to trigger transformations. They reorganize, merge, and share code across DS operators, operands, expressions, and loops and do so at a level unknown in today's optimizing compilers. Because today's compilers are working at too low a level of representation.

These tags use knowledge that is not easily determined from AST patterns and therefore, is not easily dealt with by conventional pattern-directed transforms. (e.g., knowing that an if condition will be independent of a loop control variable that will be generated). Transformations that are triggered by the tags directly re-organize the program in ways that would be hard for pattern-directed transformations to discover. They would first have to perform some very hard analysis (e.g., data flow and alias) and then have to discover a long chain of transformational miracles that choose the "right" transformation out of many choices at exactly the "right" time in the chain. This is why conventional approaches to optimizations beyond those performed by optimizing compilers engender exponential solution spaces.

The tags represent a distributed optimization plan that is created in the application domain space (largely), attached to specific pieces of the AST and executed in the programming language domain. The tags behave like little local interrupts that are waiting on certain local optimization events (e.g., substitutionofme, simplificationofme, migrationofme, etc.) or global (i.e., broadcast) events (e.g., the event signalling the start of the next translation stage).

Rather than forcing all operations into a single rule-oriented operational form (e.g., "lhs=> rhs"), the AOG generation tools are specialized to the task at hand. ("To a hammer, all problems look like nails.") AOG uses narrowly purposeful tools where it makes sense. For example, there is a partial evaluator which is invoked from the transformations when ever they reorganize the AST. Similarly, a rule engine extension to the schema language is invoke to perform simple, narrowly construed inferences (e.g., for incorporating branching logic in a loop body into the loop control).

As a consequence of the anticipatory optimization implemented via the tags, the resultant code is high performance without engendering huge search spaces.



## Reorganizing Generators

### ■ Benefits:

- ◆ Inter-component optimizations (reweavings)
- ◆ Small search space = reasonably fast

### ■ Shortcomings:

- ◆ Technology immature
- ◆ Unclear how far it can be pushed

#### **Technical basis –**

Piece part representation = domain language-based entities

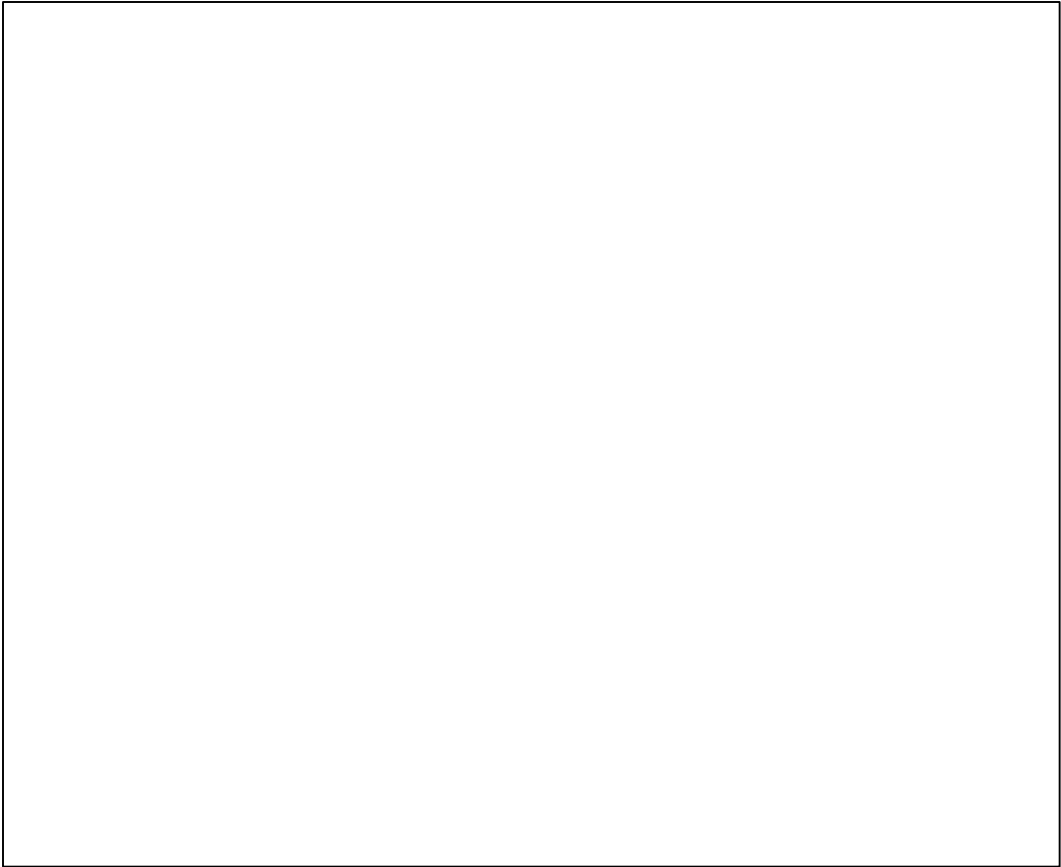
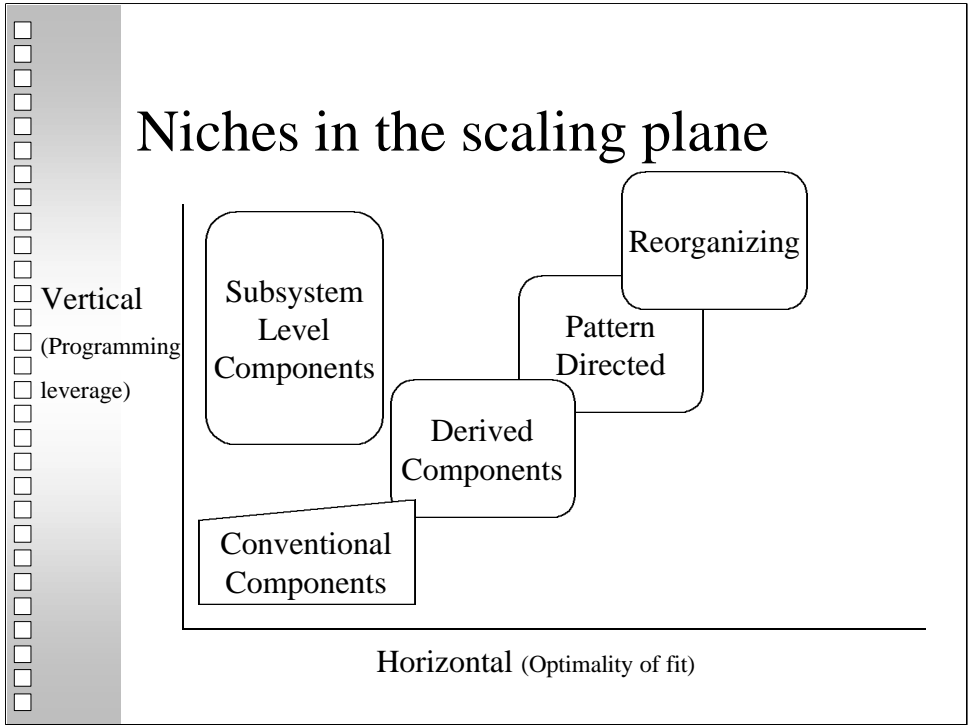
Assembly mechanism = translation and merging of language-based piece parts via pattern-based mechanisms

Optimization or reorganization mechanism = tag-directed mechanisms mitigate the inefficiencies induced by the pattern-directed paradigm and allow reorganization and reweaving via without search space explosion.

## Essence of The Technology

<u>Class</u>	<u>Elements</u>	<u>Operations</u>
Concrete Reuse	PL Struct.	Hand Assem.
Composition	Abstract PL	Inlining
PD Generator	DSL Struct.	PD Xforms
Reorg Generator	Tagged DSL	PDX & TDX







## Reuse Technologies

- Concrete components
  - ✦ e.g., Functions, OOP, Frameworks, DCOM, ...
- Compositionally Derived Components
  - ✦ e.g., Templates, GenVoca, ...
- Pattern-directed transformation systems
  - ✦ e.g., Draco, CAPE, IP, DMS, ...
- Reorganizing generator systems
  - ◆ AO generator, AOP, ...
- Inference-driven generator systems
  - ◆ Kids, Synapse, ...



## Inference-Based Generators

- Paradigm: Schema + rules of inference
- Example: Divide-and-conquer schema
  - ◆ Broad framework for solution
- User specification: Formal specification (e.g., predicate calculus)

Schemas more abstract than the Draco or AOG schema. Divide and conquer is a method by which to derive an algorithmic skeleton whereas Draco and AOG use concrete skeletons for their schemas.

Inference-based generators assemble components by logical fitting rather than structural fitting.

Of course to do this, one has to have a complete, formal specification of the program.

### **Technical basis –**

Piece part representation = derived abstractions (I.e., the schema is not a concrete a priori structure but is custom derived based on the nature of the specification).

Assembly mechanism = translation and merging of language-based piece parts via inference processes, not connecting concrete structures but custom deriving the connective plumbing by inference

Optimization or reorganization mechanism = optimization (to the degree it can be done practically) is inherent to the inference rules (I.e., intimately integrated into the inference rules for creating, deriving and integrating the schemas)



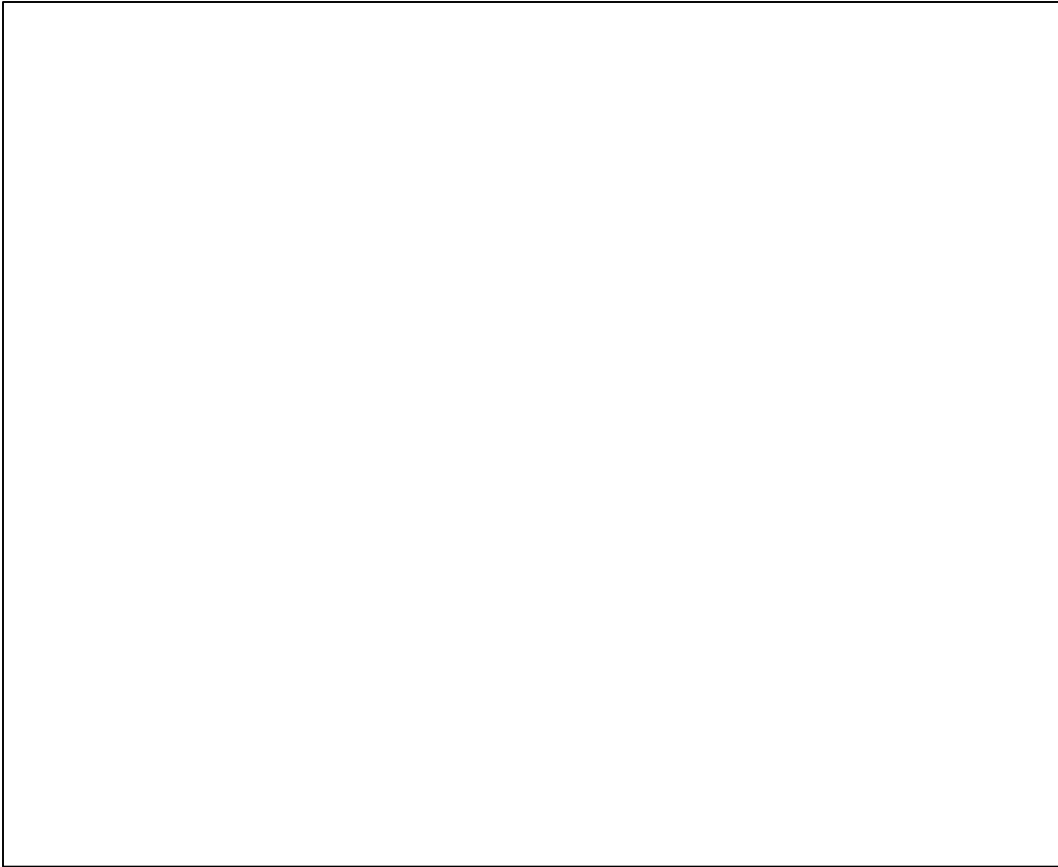
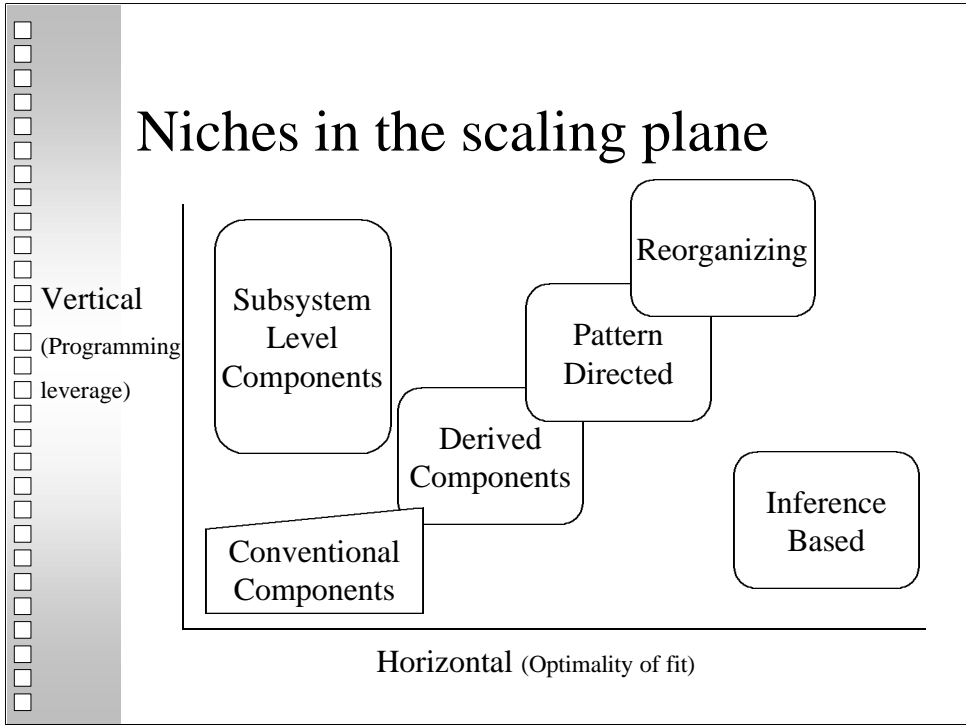
## Inference-Based Generators

- Example: Kids generator
- Benefits:
  - ◆ Highest levels of horizontal scaling
- Shortcomings:
  - ◆ Formal specifications require ultra-stable application
  - ◆ Requires well-understood domain with deep theory
  - ◆ Immaturity and narrow => low vertical scaling

Low vertical scaling because the programming leverage is very high but it is restricted to a very narrow domain. Thus, this is more like a point solution. If your domain is right on the point, you get orders of magnitude of leverage for that point and effectively zero outside of that small point region.

## Essence of The Technology

<u>Class</u>	<u>Elements</u>	<u>Operations</u>
Concrete Reuse	PL Struct.	Hand Assem.
Composition	Abstract PL	Inlining
PD Generator	DSL Struct.	PD Xforms
Reorg Generator	Tagged DSL	PDX & TDX
Infer Generator	DSL+Logic	Inference





## Reuse Technologies

- **Concrete components**
  - ✦ Limited islands of successful reuse
- **Compositionally Derived Components**
  - ✦ More horizontal scaling but no inter-part reweaving
- **Pattern-directed transformation systems**
  - ✦ Extends both but big search space for reweaving
- **Reorganizing generator systems**
  - ✦ Reweavings gain horiz. scaling but immature
- **Inference-driven generator systems**
  - ✦ Greatest horiz. scaling but formality limits use



## References

- Biggerstaff, A Perspective of Generative Reuse, *Annals of Software Eng.*, 1998
- Biggerstaff, Fixing Some Transformation Problems, *Proc. Of Automated Software Engineering*, 1999.
- Batory et al, Scalable Software Libraries, *Foundations of SE*, 1993
- Neighbors, Draco: A Method for Engineering Reusable Software Systems, in *Software Reusability*, 1989.



