To appear. Invited presentation at 12th International Workshop on Program Comprehension IWPC 2004 June 24-26, 2004, Bari, Italy.

Program Comprehension in Generative Programming: A History of Grand Challenges

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

Abstract

The communities of *Generative Programming (GP)* and *Program Comprehension (PC)* look at similar problems: GP derives a program from a specification, PC derives a specification from a program. A basic difference between the two is GP's use of specific knowledge representations and mental models that are essential for program synthesis. In this paper, I present a historical review of the Grand Challenges, results, and outlook for GP as they pertain to PC.

1 Introduction

I have worked in the areas of generative programming, product-lines, domain-specific languages, and component-based architectures for over twenty years. The emphasis of my work has been on program synthesis and design automation. The importance of these problems is intuitive: through automation we can achieve better productivity, increase software quality, reduce maintenance costs, and improve program comprehension.

Generative Programming (GP) is an automated process that maps a specification — ideally a declarative specification — to an executable. *Program Comprehension (PC)* is the complement: it is not an automated process whose goal is to reconstruct a specification for an executable [16]. The core approaches of GP and PC are understandably similar. Both use domain models to express architectural relationships among application components, both use situational models to flesh out architectural designs with algorithmic details, and both use program models to express programmatic and language details [16][21][13].

GP and PC ask similar questions. The PC community applies techniques to understand a program in sufficient detail in order to manually customize, repair, or extend the program correctly. The GP community starts with a program domain — a family of related programs — and applies techniques to understand the domain in sufficient detail in order to automatically synthesize programs in that domain and to mechanize certain maintenance tasks. If there is a fundamental difference between PC and GP, it is that GP requires very specific models of knowledge representation in order to automate the design and synthesis of software.

As a rule, different communities think differently about similar problems. They have different formulations, different starting points, and different goals. To appreciate a discipline requires one to know key events in its history: why study *these* problems? Why *these* solutions?

In this paper, I present an historical review of concept development in GP as it pertains to knowledge representation and PC. This history is intimately related to software modularity (knowledge chunking) and the quest for understanding the fundamental structures and organizations of programmatic knowledge.

What's so special about GP knowledge representation that it deserves such attention? Answer: Software design is an artform, and as long as it remains so, our abilities to automate software design and to simplify tasks of program comprehension will be inherently limited. A "Science of Software Design" is sorely needed.

Scientific theories arise by studying groups of related phenomona. Theories of atomic physics, for example, were not created by studying one kind of atom, but all kinds. Similarly, astronomical theories are not created by studying one star or one galaxy, but many stars and galaxies. In short, scientists gain fundamental insights into nature by creating theories that explain variations in related phenomena.

I argue that GP is leading us towards a "Science of Software Design". GP models are predictive and constructive theories of how programs in a domain work and can be synthesized. These theories rely on an unusual form of knowledge representation. How one comprehends a generated program may be very different from how one comprehends a hand-written program.

Interestingly, the key results in GP did not occur in chronological order. In the following sections, I summarize contributions in an order that I think makes their significance easier to grasp and appreciate.

2 Design Maintenance

In 1992, Ira Baxter observed that the software engineering community was attacking the problem of program maintenance in the wrong way [8][1]. He noted that engineers are burdened by the task of *code maintenance*. That is, they are given a program with the task of extending or maintaining it where the key details of the program's design, rationale, and abstractions have been lost. That is, the key information needed for correct program maintenance and extension is gone and must be reconstructed. (These, of course, are the key problems of program comprehension).

Baxter advocated that instead of code maintenance, software engineers should perform *design maintenance (DM)*. That is, we should maintain the design of the program (which presumably is a smaller document that is much easier to understand) and derive the program's code automatically. He advocated that a program transformation and AI planning approach could be used to realize DM.

The *transform* is the basic unit of modularization. It is a function that maps a program to a program. The application of a transform at a particular location is a *transformation*, and is denoted by:

`pattern1` $\Rightarrow$ **`pattern2 @ location`**

That is, code **`pattern1`** is replaced by code **`pattern2`** at a designated **`location`** in a program. Figure 1a illustrates a transform that distributes multiplication over addition. Figure 1b shows a program (in parse tree format) before the application of the transform at node **`<1>`** and Figure 1c is the result.

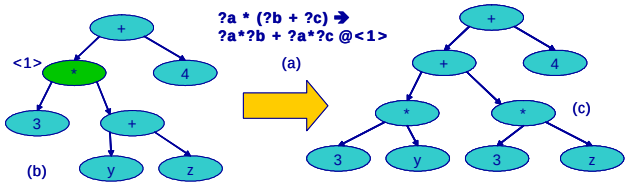

Figure 1. A Program Transform

Given the use of transforms, the core ideas of design maintenance unfolds (Figure 2). One begins with a specification of a program (**`f0`**) and applies a series of transformations to produce the target program (**`fn`**). The transforms that are used are not randomly chosen or haphazardly applied, but rather selected on the basis of performance and functionality goals (or "how to implement" goals) that are part of the original specification. A program's *design history* is recorded by saving the rationale for each decision point (indicated by the **`Gi`** nodes in Figure 2).

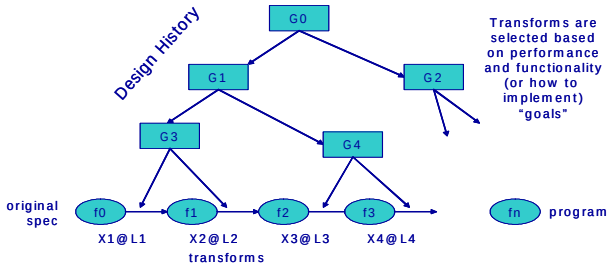

Figure 2. Design History

A design history records what was desired (the original specification), how it was achieved (the series of transformations that were applied), and why this particular implementation (the justification for these transformations). Given a design history, it would seem we have all the information about a program's development!

Baxter also conjectured that the generation of program design histories might be expensive. If a change is made to the specification, the recorded history could be used to replay the design decisions that were unaffected. A new program would be generated by filling in the missing pieces caused by the change.

While it was not immediately obvious, the concept of Design Maintenance raised seven Grand Challenge Problems, which I have come to believe are the Holy Grail(s) of Generative Programming:

#1: Language Challenge: In what *language* should a program specification be written? First order logic is too sophisticated for most people. Ideally, we want a language to be declarative — allowing users to specify what they want, not how to do it. Further, we want a language that people without advanced technical degrees can write and understand.

#2 Automatic Programming Challenge: Mapping a declarative specification to an efficient executable is *hard*. Called *Automatic Programming (AP)*, all but the most pioneering researchers gave up on AP in the early 1980s, as the techniques that were available then did not scale [1]. Design Maintenance requires AP to be solved.

#3 Consistent Refinement Challenge: Individual transforms need not be correctness preserving. Rather, the net effect of applying a *set* of transformations must be correctness preserving. This is similar to database transactions. Each tuple update moves the database through an inconsistent state; only after all of the updates have been applied is the database consistent. The hallmark of transactions is atomicity: either all of its changes are made, or none are. Anything less would corrupt the database. The same applies here: either the entire set of transformations is applied, or none are. This is the problem of *consistent refinement*: there are many places in a program that must be transformed in a

1. Arguably the CASE community much earlier advanced the notion of updating designs; Baxter gets credit for saying it in a brazen, clear way [9].

coordinated way to preserve program consistency. How can consistent refinements be realized?

#4 Modularity Challenge: The unit of knowledge representation (transforms) seems too small. The progress of programming languages and software design is marked by increasing levels of abstraction: functions, to classes, to packages, to components, etc. Transforms buck the trend.

Creating a knowledge base of transforms is possible, but when and where should transforms be applied? How should the knowledge base of transforms be searched? And the space of possible programs that could be generated is gargantuan: how can this space be enumerated, let alone searched, efficiently?

#5 Design History Challenge: From a program comprehension viewpoint, DM is great stuff: we point at a piece of code and immediately know its derivation. But how can human-understandable explanations of a derivation be produced automatically? How can tens of thousands (or more) of miniscule steps be abstracted into a human-comprehendible form? How is it practical to maintain the history of decisions for later partial replay?

#6 Scale Challenge: How can DM scale to large systems? It only works for small systems, DM should be abandoned. And finally,

#7 Science of Design Challenge: What is a scientific or mathematical basis for all this? What is a "Science of Software Design"?

DM sets the stage for our next major result, which ironically appeared twelve years *prior* to the DM paper…

3 Neighbor's Draco

Jim Neighbors' 1980 Ph.D. explained the importance of levels of abstraction, *domain-specific languages (DSLs)*, and domain-specific optimizations in program generation [17]. Neighbors also advocated a program transformation (and maybe AI planning) approach to program generation, but this is not surprising as both he and Baxter worked under Peter Freeman at U.C. Irvine.

Figure 3 illustrates the Draco paradigm[2]. Neighbors advocated that programs should be written in DSLs, because doing so, they have their most compact and elegant expression. When a DSL program is initially written, it is unoptimized. Domain-specific optimizations can be applied to an unoptimized DSL program to map it to an optimized DSL program. This is possible because domain abstractions are still visible. Or stated another way, *one cannot optimize abstractions that have been compiled away.*

2. In fact, the Draco paradigm applies to any language, not just DSLs.

An optimized DSL program is then translated to an equivalent program in a more primitive DSL that exposes lower-level abstractions. This program is unoptimized w.r.t. these abstractions, and the process of mapping it to an optimized DSL program and then translating it to an equivalent program in an even more primitive DSL is performed. The process repeats until machine code is reached.

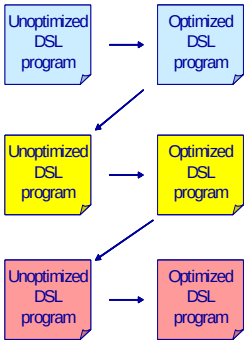

Figure 3. Draco Paradigm

As an artificial example, consider a program that uses state machines to perform file updates. The program is written in a DSL for state machines. A state machine optimization might, for example, eliminate unreachable states or fuse states that are connected by null transitions. An optimized state machine program could be translated into a language with explicit support for file operations. A possible optimization on files is replacing "`open(file); close(file);`" statements with no-ops. An optimized file DSL program could then be mapped to a general purpose language for further optimization and translation.

Draco provided two valuable clues to the Modularity Challenge. First, transforms for program optimization should be separated from those of program generation. Design optimization corresponds to the horizontal translations in Figure 3, while program generation corresponds to the vertical translations. Second, applying a set of miniscule transforms is equivalent to applying a single large-scale transform. The Draco paradigm is an algebraic recurrence relation **(1)** that optimizes a $\texttt{program}_i$ in language $\texttt{L}_i$ and generates an equivalent $\texttt{program}_{i-1}$ in a lower-level language $\texttt{L}_{i-1}$:

$$\texttt{program}_{i-1} = \texttt{generate}_i(\ \texttt{optimize}_i(\ \texttt{program}_i\ ))\quad (1)$$

Further, Draco provided a clue for the Language Challenge: a language for the masses will likely be some sort of DSL.

There are, of course, open questions such as: how generally applicable is Draco? Programs have many levels of abstractions. Programmers use Java interfaces, for example, to define these levels in their programs and neither DSLs or optimizations are needed. So when and where are DSLs — or more generally *different languages* — needed?

Draco, unfortunately, does not bring us closer to surmounting the Design History Challenge. Optimizations make program comprehension fundamentally harder. A fragment of Java code may be the result of optimizations at multiple levels of abstraction. To understand this code would involve transforming the original specification; optimizations often break clean encapsulations obscuring specifica-

tion simplicity. With optimizations, the need to maintain a Design History becomes that much more important, and apparently that much more difficult.

Our next result appeared ten years after Draco…

4 Feature Oriented Domain Analysis (FODA)

Kyo Kang was working at the SEI in 1990 when he made a fundamental contribution to the understanding of product-lines [14]. The motivating problem was clear enough: successful programs spawn variants. Designing and building each variant from scratch was way too expensive (even in those days of rampant reinvention). He and his colleagues realized that a more practical approach would be to create a design for a family (or domain) of programs that leverages common assets. This family of programs was called a *product-line*. The fundamental contribution was the recognition that features were the primary means by which members of a product-line were distinguished.

To appreciate features, consider the following thought experiment. How do you describe a program to someone? Hopefully you wouldn't say which DLLs or widgets the program used. Instead, you are more likely to describe the program by saying what *features* the program offers it clients. Abstractly, you might say that `program1` has features `X`, `Y`, and `Z`, while a competing `program2` has features `X`, `Q`, and `R`. The reason why you describe programs in this way is because features align with requirements — or features *are* the requirements. Thus, *programs can be specified as compositions of features.* Feature-specifications of products are common, it is just not common for software (yet).

As an example, go to the Dell or Gateway web site. You'll find lots of HTML pages that are *declarative DSLs* — using check boxes, combo-boxes, and radio-buttons, a client can declaratively specify features or constraints on features for a desired personal computer. Filling in these specifications is simple; the hard part is writing the cheque! We want to do the same for software.

In retrospect, FODA provided a solution to the Language Challenge: declarative feature specifications can be written and understood by people that do not have advanced technical degrees. So by creating feature models of product-lines, we have an attractive and intuitive answer to the Language Challenge.

FODA also leads us toward the solution of the Modularity Challenge. Features define a new form of modularity — they are a higher-level, more abstract way in which to understand programs. It was soon realized that feature implementations often do not align along class and package boundaries. For example, program `P` in Figure 4 consists of four packages `A`—`D`. Adding feature `F` requires changes in all of these packages.

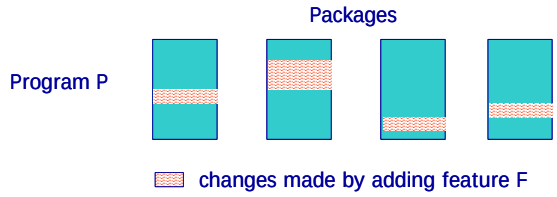

Figure 4. Consistent Refinement Problem Revisited

Astute readers may recognize Figure 4 as an example of "cross-cuts" or "aspects". This would be correct, but more specifically, Figure 4 is a classical example of the consistent refinement problem. That is, Figure 4 shows there are multiple places in a program that must be transformed (or updated) in a coordinated way to preserve program consistency. Either all of these changes to `P` are made, or none are. Anything less will leave `P` in a corrupted state. Thus, if we could figure out how to modularize features, we could crack both the Consistent Refinement Challenge and the Modularity Challenge.

Features also contributed in another way to a better understanding of the Modularity Challenge. Kang, myself, and others used preprocessor `#ifdef`—`#endif` statements to surround feature-specific code fragments. System-generation (a.k.a. *sysgen*) approaches were employed, where the code for a particular feature was included in a target system if certain preprocessor flags were set. This implementation technique helped broaden the focus of the program generation community from the program transformation and AI planning world to include main-stream programming techniques.[3] After all, preprocessors and sysgen techniques were known for decades. And this provided a good historical tie-in with the pioneering work in the late 1980s and early 90's of Craig Cleaveland [11] and Paul Bassett [3] on their use of preprocessor-based program generator technologies. (I am sure that Cleaveland and Bassett were wrestling with the GP Grand Challenges too, in one way or another).

The use of preprocessors led to an important cross-roads in the GP community. The next significant advances occurred about eight years after FODA…

5 Programming Language Representations of Transforms

Every *programming language (PL)* is a knowledge representation language. PLs like Java, C++, C#, etc. have formal and precise definitions of programming concepts and constructs. Preprocessors, on the other hand, are ad hoc

3. Or more accurately, there were two different program generation communities — those using program transformations and those using preprocessors — that slowly began to merge. This was a primary motivation for creating the *International Conference on Software Reuse (ICSR)*.

extensions to programming languages. They allow us to express concepts that cannot (or are hard to) express directly in a PL itself. In effect, preprocessors allow us to express concepts that hopefully will appear in future languages. It's not surprising, therefore, to see that generator technologies have often been preprocessor-based. There have been movements in the broader GP community to give PL support for generative programming concepts, such as template metaprogramming [22].

Recall Baxter's definition of a transformation:

pattern1 $\Rightarrow$ **pattern2 @ location**

Suppose **pattern1** is a method signature (like "**int five()**") and **pattern2** is a method body (like "**return 5;**". Their combination is a *method definition*:

```
int five() { return 5; }
```

What's missing is the **location**. But this is implicit: any place in a program that invokes **five()** is the location at which the **five()** transform can be applied. An application of a method transform is called *inlining* — the body of the method replaces its call.

We can take this analogy further: if a method is a transform, then an object-oriented class encapsulates a set of consistent transforms! That is, all methods (transforms) in a class are designed to work together. So Figure 5a is a class (set of consistent transforms) that define a container implementation, Figure 5b is a program that invokes these methods, and Figure 5c is an inlined version of this program using the container class of Figure 5a. This idea scales to packages and larger entities: they can be viewed as progressively larger sets of consistent transforms.

```
class container {
  void getFirst() {
     ..getFirstAlg..
  }
  void getNext() {
    ..getNextAlg..
  }
}
```
(a)

```
// main program
container x;
...
x.getFirst();
...
x.getNext();
...
```
(b)

```
// main program
container x;
...
..getFirstAlg..
...
..getNextAlg...
...
```
(c)

Figure 5. Class-Scale Transform Sets

Of course, one can't always do such inlining. Polymorphism often precludes inlining because the exact type of object that is invoked in a piece of code is not known (i.e., it could be any object belonging to a type or any of its subtypes, where subtypes override method definitions).

However, there are common situations in the implementations of features where inlining methods and entire classes are exactly what is needed. The idea is called in-place extensions of classes, or "inheritance without inheritance hierarchies". The following are ideas that have been used to great effect in our work at the University of Texas [7].

Consider Figure 6a which defines class **K**. Figure 6b defines a refinement or extension of **K**. That is, method **foo()** of Figure 6b extends method **foo()** of Figure 6a by inlining the original definition (that's what **Super.foo()** means) followed by statement **c**. Similarly, method **bar()** is extended by statement **d** followed by the original body of **bar()**. The member declaration "**int f;**" means add member **f** to class **K**. Figure 6c is the result. Readers may recognize that Figure 6c is equivalent to having the class of Figure 6b be the subclass of Figure 6a, and their inheritance hierarchy squashed into one class. As mentioned earlier, we can scale this concept so that many classes are refined (extended) simultaneously. This is how we implement features and define them in a programming language.

Stated algebraically, if **K** denotes Figure 6a, and **R** defines the refinement of **K** as in Figure 6b, the class of Figure 6c is the result **R•K**, where • is a class composition operator. More on this later.

```
class K {
 void foo(){
   a;b;
 }
 void bar(){
  e;
 }
}
```
(a)

```
refines class K {
 void foo(){
   Super.foo();c;
 }
 void bar(){
  d; Super.bar();
 }
 int f;
}
```
(b)

```
class K {
 void foo(){
   a;b;c
 }
 void bar(){
  d;e;
 }
 int f;
}
```
(c)

Figure 6. Class Extension in AHEAD

Recall the Consistent Refinement Challenge: there are many places in a program that must be transformed in a coordinated way to preserve program consistency. The class refinement/extension in Figure 6 is an effective way to solve the consistent refinement problem.

Our last programming language example is Aspect Oriented Programming. AspectJ introduces an important kind of transformation to Java [15]. The definition of a dynamic cross cut is:

pointcut_predicate $\Rightarrow$ **advice**

It's meaning is simple: A *join-point* is a designated point in a program (e.g., method invocation, initialization body, catch body, etc.). A **pointcut** is a predicate that defines a set of join-points and **advice** is a code pattern that is to be added at each qualified join-point location. I hope that the transformation heritage of AspectJ cross-cuts is easy to see. An aspect is a set of such cross-cut definitions (i.e., a set of consistent transforms).

Let's see how advances in programming languages impacts the Design History Challenge. If programming languages add constructs to implement feature modularities, then high-level program abstractions will be explicit in program source code. That is, semantic units of functionality and their implementations will be more clearly separated. With such advances, monolithic programs with code fragments

of multiple features that are "scattered and tangled" should be relics of the past. Thus, it seems that a useful step toward improved program comprehension and the Design History Challenge is to make architectural (or high-level) design concepts more explicit in programming languages and in program source.

Now let's summarize what we've seen so far. The GP dream is to realize Design Maintenance. We have made significant progress on:

- Language Challenge — use declarative feature-based specification languages
- Consistent Refinement Challenge and the Modularity Challenge — extend programming languages to express sets of consistent transforms

We still do not yet know how to solve:

- Automatic Programming Challenge
- Design History Challenge
- Scalability Challenge
- Science of Design Challenge

Our next contribution is arguably the most significant result in GP. Ironically, it occurred 25 years ago, even before Draco. It was conceived for a significant domain, had a revolutionary impact on industry, and occurred around the time researchers gave up on automatic programming!

6 Relational Query Optimization

Here's how relational query optimization works [18]: An SQL statement is parsed into an inefficient relational algebra expression. A query optimizer rewrites the expression into a semantically equivalent expression that has better performance characteristics. A code generator translates the optimized expression into an efficient program.

SQL is a classic example of a declarative DSL. It is a language that is specific to tabular representations of data. The code generator, which maps a relational algebra expression to an executable, is an early example of GP. The query optimizer is the key to Automatic Programming: it searches the space of semantically equivalent expressions to locate an expression which has good (or optimal) performance characteristics.

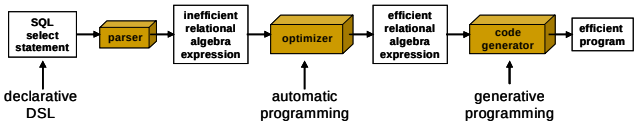

Figure 7. Relational Query Optimization

Let's look at what relational database researchers accomplished. First, they created a practical form of Design Maintenance! That is, programmers maintain declarative SQL specifications, *not* the code that is generated from these specifications. This approach has withstood the test of time and has been a *big* win.

Second, they stated a practical formulation of the Automatic Programming problem and solved it!

Third, a high-level design history is simple to maintain: remember the original SQL statement, and maybe the optimized relational algebra expression. Given these two as a road-map, a programmer has a lot of information to comprehend a generated query evaluation program.

Fourth, it is interesting to see that the *Relational Query Optimization (RQO)* paradigm fits beautifully with Draco. There is a clear use of multiple languages, domain-specific optimizations, and multiple levels of abstraction. This is yet another clue to the Scalability Challenge.

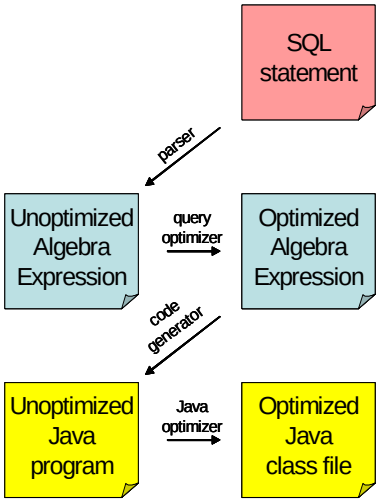

Figure 8. RQO and Draco

Fifth and most importantly, RQO provides an invaluable clue to the Science of Design Challenge. What relational database researchers achieved is nothing short of remarkable. They automated the development of query evaluation programs: these programs were hard to write, hard to optimize, and even harder to maintain. They created an *algebra-based science* to specify and optimize query evaluation programs. They accomplished this by identifying fundamental operators of this domain (relational algebra), the design of programs in this domain were represented by expressions (compositions of relational operators), and identities among these operators were used to optimize expressions (program designs). That is quite a lot!

Several open problems remain: how do features relate to RQO? And how does the RQO paradigm scale to large programs? Let's move forward to the mid-1990s to some work that I did that answers these questions and ties together more loose ends. (Incidentally, at the time that I was doing this work, I *definitely* was not cognizant of the history I am unfolding in this paper).

7 Early FOP and GenVoca

The 1992 paper on GenVoca defined a simple, mathematical model of what is now called *Feature-Oriented Programming (FOP)* [4]. A GenVoca model of a domain is a

set of operators that defines an algebra. Each operator implements a feature. We write:

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

to mean model `M` has operators (or features) `f`, `h`, `i`, and `j`. One or more of these operators are *constants* that represent base programs:

```
f       // a program with feature f
h       // a program with feature h
```

The remaining operators are *functions* or *large scale transforms* which represent program extensions:

```
i(x)    // adds feature i to program x
j(x)    // adds feature j to program x
```

The design of an application is named composition of operators called an *equation*:

```
prog1 = i•f     // program w. features i and f
prog2 = j•h     // program w. features j and h
prog3 = i•j•h   // program w. features i,j,h
```

where • denotes function composition. The family of programs that can be created from a model is it's *product-line*.

A GenVoca expression represents the *design* of a program. Such expressions (and hence program designs) can be automatically optimized. This is possible because a function represents both a feature *and* its implementation. That is, there can be different functions with different implementations of the *same* feature:

```
k1(x)   // adds k with implementation1 to x
k2(x)   // adds k with implementation2 to x
```

When an application requires the use of feature `k`, it is a problem of *expression optimization* to determine which implementation of `k` is best (e.g., provides the best performance). Of course, more complicated rewrite rules can be used. Thus, it is possible to design efficient software automatically (i.e., find an expression that optimizes some criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning — which is exactly the counterpart to relational query optimization — is [6].

The program synthesis paradigm of GenVoca is illustrated in Figure 9. Program `P` consists of four classes `c1`—`c4`. It is synthesized by composing features `f1`, `f2`, and `f3`. `f1` encapsulates a fragment of classes `c1`—`c3`. `f2` extends each of these classes and introduces `c4`. `f3` extends all four classes. The manner in which these classes are defined and extended was described earlier regarding the discussion of Figure 6. Feature (functions) are implemented by sets of consistent transforms.

The significance of GenVoca is two-fold. First, it shows the connection among features, algebras, consistent refinements, and feature modularities. A feature is synonymous

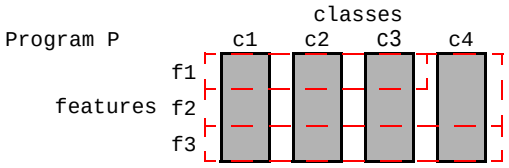

Figure 9. Program `P = f1 • f2 • f3`

with a large-scale transform. Second, it allows medium-sized programs (in excess of 80K LOC) to be synthesized. We and others have synthesized database systems in 1987 [4], network protocols in 1989 [4], avionics in 1994 [5], extensible compilers in 1998 [12], and program verification tools in 2001 [20], among others, using these ideas.

We now have a preliminary solution for each of the Grand Challenges:

- Language Challenge: use declarative feature DSLs
- Automatic Programming: extend RQO paradigm
- Consistent Refinement and Modularity: extend PLs to express sets of consistent transforms
- Design History: remember original spec and its optimized GenVoca expression
- Scalability Challenge: medium-scale program synthesis
- Science of Design: algebraic foundation for program synthesis

The next result revolutionized my thinking on program generation and program comprehension. It helps answer the nagging questions of scale (how can we scale design maintenance to even larger programs?) and the Draco refrain: how do different languages fit into the PC and GP worlds?

8 Recent FOP and AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) is the successor to GenVoca [7]. AHEAD shows how feature models scale to the synthesis of multiple programs and multiple representations, and that software design has an elegant algebraic structure that is expressible as nested sets of expressions. The following sketches the basic ideas.

8.1 Multiple Program Representations

Today's systems are not individual programs but groups of collaborating programs such as client-servers and tool suites of integrated development environments. Further, systems themselves are not solely defined by source code. Architects routinely use many knowledge representations to express *and understand* a system's design, such as process models, UML models, makefiles, formal specifications, etc. If we are to understand and generate these larger systems, we cannot solely focus on code comprehension and code synthesis. We must understand programs from a much broader knowledge representation viewpoint.

That a program has many representations is reminiscent of Platonic forms. That is, a program is a form. Shining a light on this program casts a shadow that defines a representation of that program in a particular language. Different light positions cast different shadows, exposing different details or *representations* of that program. For example, one shadow might reveal a program's representation in Java, another an HTML document (which might be a design document). There are class file or binary representations of a program, makefile representations, performance models, an so on. Further, we want to encapsulate all of these representations.

Consider a compiler **J**. It uses (at least) two representations: $\mathbf{Code}_J$ and $\mathbf{Gram}_J$. $\mathbf{Code}_J$ defines a set of Java classes that implement the body of the compiler, and $\mathbf{Gram}_J$ is a grammar from which **J**'s parser is derived. We say that **J** encapsulates $\mathbf{Code}_J$ and $\mathbf{Gram}_J$ and write this relationship as:

$$\mathbf{J} = \{ \mathbf{Code}_J, \mathbf{Gram}_J \}$$

Set notation denotes encapsulation.

8.2 Generalize Transforms

Adding a new feature to a program changes any or all of its representations. For example, if a new feature **F** is added to a program, we would expect changes in the program's code (to implement **F**), documentation (to document **F**), makefiles (to build **F**), formal properties (to characterize **F**), performance models (to profile **F**), and so on.

For example, suppose feature **F** adds state machine declarations to the language for which **J** is a compiler. **F** would change the code and grammar artifacts of **J**. Let $\mathbf{Code}_F$ and $\mathbf{Gram}_F$ denote these changes. We say that **F** encapsulates $\mathbf{Code}_F$ and $\mathbf{Gram}_F$ and write this relationship as:

$$\mathbf{F} = \{ \mathbf{Code}_F, \mathbf{Gram}_F \}$$

8.3 Generalize Composition

Given **F** and **J**, how do we compute **F•J**? The answer: we expand the definitions for **F** and **J** and compose their corresponding representations:

$$\begin{aligned} \mathbf{F \bullet J} &= \{ \mathbf{Code}_F, \mathbf{Gram}_F \} \bullet \{ \mathbf{Code}_J, \mathbf{Gram}_J \} \\ &= \{ \mathbf{Code}_F \bullet \mathbf{Code}_J, \mathbf{Gram}_F \bullet \mathbf{Gram}_J \} \qquad (2) \end{aligned}$$

That is, the grammar artifact of **F•J** is the original grammar artifact, $\mathbf{Gram}_J$, composed with its changes, $\mathbf{Gram}_F$. Similarly, the code artifact of **F•J** is the original code artifact, $\mathbf{Java}_J$, composed with its changes, $\mathbf{Java}_F$. In effect, equation **(2)** defines an algebraic Law of Composition: *it tells us how composition distributes over encapsulation.* A more general definition of this law is presented in [7], where any set of artifacts can be encapsulated and extended.

We've seen this idea before. Recall Figure 4 and Figure 9. Both of these figures illustrate that a program contains multiple artifacts (four artifacts specifically), and that adding a feature updates some or all of these artifacts. **(2)** *is an algebraic statement of consistent refinement*: program **J** encapsulates a set of artifacts and feature **F** encapsulates a set of changes to these artifacts. When **F** is applied to **P**, all of **F**'s changes are applied to **P**.

8.4 Generalize Modularity

A *module* is a containment hierarchy of related artifacts. Figure 10a shows that a class is a 2-level containment hierarchy that encapsulates a set of methods and fields. An interface is also a 2-level containment hierarchy that encapsulates a set of methods and constants. A package is a 3-level containment hierarchy encapsulating a set of classes and interfaces. A J2EE EAR file is a 4-level hierarchy that encapsulates a set of packages, deployment descriptors, and HTML files.

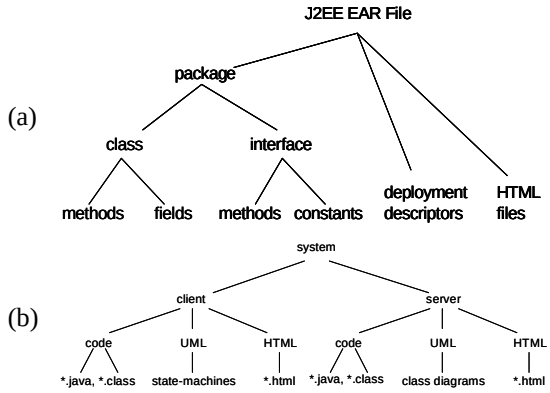

Figure 10. Modules are Containment Hierarchies

In general, a module hierarchy can be of arbitrary depth and can contain arbitrary artifacts. This enables us to define a module that encapsulates multiple programs. Figure 10b shows a system to encapsulate two programs, a client and a server. Both programs have code, UML, and HTML representations with sub-representations (e.g., code has Java files and binary class files, UML has state machines and class diagrams). Thus, a module allows us to encapsulate all needed representations of a system.

Module hierarchies have a simple algebraic representation as nested sets of constants and functions. Figure 11a shows package **K** encapsulating **class1** and **class2**. Similarly, **class1** encapsulates method **mth1** and field **fld1**, and **class2** encapsulates **mth2** and **mth3**. The corresponding set notation is shown in Figure 11b.

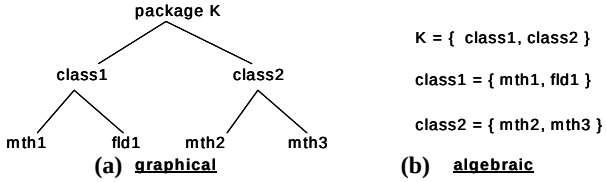

Figure 11. Modules and Nested Sets

8.5 Generalize GenVoca

A GenVoca model is a set of constants and functions. An AHEAD model is also a set of constants and functions, but now a constant represents a hierarchy that encapsulates different representations of a base program. An AHEAD function or large-scale transform is a hierarchy of extensions — that is, it is a containment hierarchy that can add new artifacts (e.g., new Java and HTML files) at various points in a target hierarchy, and can also refine/extend existing artifacts. When features are composed, corresponding program representations are composed. If the representations of each feature are consistent, then their composition is consistent. This is exactly what we want.

8.6 Implementation Details

We implement module hierarchies as directory hierarchies. Figure 12a shows our algebraic representation of a module, and Figure 12b shows its directory representation.

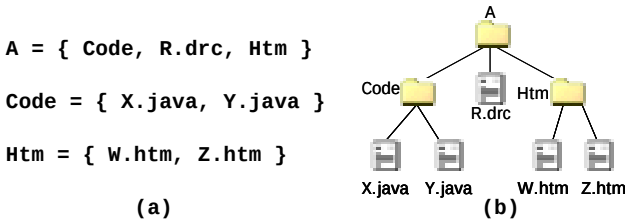

Figure 12. Algebra and Directory Correspondence

Feature composition is directory composition. That is, when we compose features, we fold their corresponding directories together to produce a directory whose structure is isomorphic to the original directories. For example, the `X.java` file of `C = B•A` in Figure 13 is produced by composing the corresponding `X.java` files of `B` and `A`.

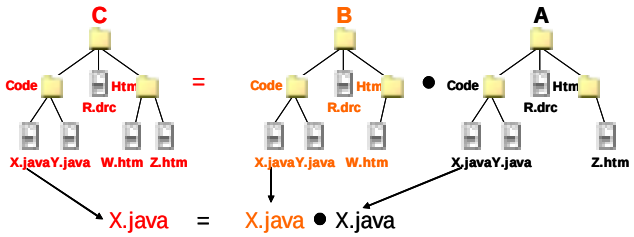

Figure 13. Composition of Feature Directories

Our implementation is driven purely by algebraic manipulation. We evaluate an expression by alternatively expanding nonterminals and applying the Law of Composition:

$$
\begin{aligned}
\texttt{C} &= \texttt{B} \bullet \texttt{A} \\
&= \{\texttt{Code}_B, \texttt{R.drc}_B, \texttt{Htm}_B\} \bullet \{\texttt{Code}_A, \texttt{R.drc}_A, \texttt{Htm}_A\} \\
&= \{\texttt{Code}_B \bullet \texttt{Code}_A,\ \texttt{R.drc}_B \bullet \texttt{R.drc}_A,\ \texttt{Htm}_B \bullet \texttt{Htm}_A\} \\
&= \{\{\texttt{X.java}_B,\ \texttt{Y.java}_B\} \bullet \{\texttt{X.java}_A,\ \texttt{Y.java}_A\}, \\
&\quad\ \texttt{R.drc}_B \bullet \texttt{R.drc}_A,\ \{\texttt{W.htm}_B\} \bullet \{\texttt{Z.htm}_A\}\ \} \\
&= \{\{\ \texttt{X.java}_B \bullet \texttt{X.java}_A,\ \texttt{Y.java}_B \bullet \texttt{Y.java}_A\ \}, \\
&\quad\ \texttt{R.drc}_B \bullet \texttt{R.drc}_A,\ \{\texttt{W.htm}_B,\ \texttt{Z.htm}_A\}\}
\end{aligned}
$$

The result is a nested set of expressions. Each expression tells us how to synthesize an artifact of the target program. That is, the `X.java` artifact of feature `C` is computed by $\texttt{X.java}_B \bullet \texttt{X.java}_A$; the `Y.java` artifact of `C` is computed by $\texttt{Y.java}_B \bullet \texttt{Y.java}_A$, the `R.drc` artifact of `C` is computed by $\texttt{R.drc}_B \bullet \texttt{R.drc}_A$, and so on. Thus, there is a simple interpretation for every computed expression, and there is a direct mapping of the nested set of expressions to the directory that is synthesized.

Figure 14 illustrates the AHEAD paradigm. An engineer defines a system by declaratively specifying the features it is to have, typically using some GUI-based DSL. The DSL compiler translates the specification into an AHEAD expression, which is then expanded and optimized, producing a nested set of expressions. Each expression is typed — expressions that synthesize Java files are distinguishable from expressions that synthesize grammar files — and is submitted to a type-specific generator to synthesize that artifact. The set of artifacts produced are consistent w.r.t. the original declarative specification. This is a scaled generalization of the Relational Query Optimization paradigm and an example of Design Maintenance.

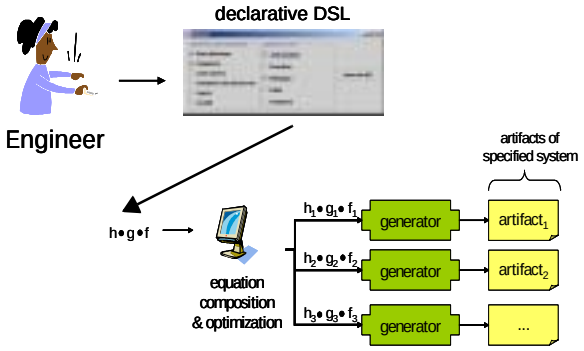

Figure 14. Program Synthesis Paradigm of AHEAD

What does AHEAD mean to the Program Comprehension community? First, there should be no worries that the core problems of PC will go away. However, it does suggest that we will be able to synthesize a variety of documents to aid program understanding. Such documents will be used by people who are manually integrating generated programs or components into some hand-crafted system. Surely, this will be better than inconsistent documents programmers have today.

However, AHEAD does not support the detailed design histories that Baxter advocated. Instead, it provides levels of abstraction for various program artifacts, where histories or code ancestries are maintained (as expressions) down to the class or method level. But this may not be enough.

Stated another way, AHEAD is an architectural model of program synthesis. Below a certain level of detail, its ability to predict or explain is minimal. Just as Newtonian mechan-

ics has a limited ability to describe atomic interactions, more sophisticated models of program generation and program comprehension (such as the work of Kestrel [10] and NASA Ames [19]) are needed. The generation of really useful explanations may hinge on this additional level of detail. While considerable progress has been made in the last 25 years towards Design Maintenance, there is still a long way to go.

9 Conclusions

Just as the structure of matter is fundamental to chemistry and physics, so too must the structure of software be fundamental to Computer Science. By structure I mean what is a module and how do modules compose to build larger modules?

Unfortunately, the structure of software is not well-understood. Software design, which is a process to define the structure of a program, is an art-form. And as long as it remains an art-form, our abilities to automate software development and to simplify tasks of program comprehension will be inherently limited.

The history of key results in Generative Programming aim to automate, and thus formalize, informal knowledge of programmers. The major thrust of the GP community, in retrospect, has been to move closer to a practical form of Design Maintenance, and in doing so, address Challenge problems that blocked our way to significant progress.

We have made progress in understanding program synthesis using transforms and expressing transforms in programming languages. We have learned that domains of programs can be expressed as algebras, where particular programs are expressions, and the operators of these algebras are large-scale transforms. And we have learned that improved program design and comprehension relies on many different program representations besides source code.

Software has an elegant algebraic structure that is easily obscured. Unfortunately, programmers are geniuses in making simple things look complicated. Our job in GP and PC is to reveal the potential simplicity of a program and to remove its accidental complexity. Eventually, there will be a "Science of Software Design" that will guide us both in program synthesis and knowledge recovery of legacy programs. Continued progress will get us there.

Acknowledgements. I gratefully acknowledge the helpful comments of Ira Baxter and Anneliese Andrews on earlier drafts of this paper.

10 References

[1] A. Andrews, S. Ghosh, E. Choi, "A Model for Understanding Software Components", *International Conference on Software Maintenance (ICSM'02)*, Montreal, Quebec, Canada.

[2] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming", *IEEE TSE*, November 1985.

[3] P. Bassett, *Framing Software Reuse: Lessons from the Real World*, Yourdon Press Computing Series, 1996.

[4] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[5] D. Batory, L. Coglianese, et al., "Creating Reference Architectures: An Example from Avionics", *Symposium on Software Reusability*, Seattle Washington, April 1995.

[6] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE TSE*, May 2000, 441-452.

[7] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement", *International Conference on Software Engineering (ICSE-2003)*.

[8] I.D. Baxter, "Design Maintenance Systems", *Communications of the ACM*, Vol. 55, No. 4 (1992) 73-89.

[9] I.D. Baxter, personal correspondence, March 2004.

[10] M. Becker, et al., "Planware II: Synthesis of Schedulers for Complex Resource Systems", Kestrel Institute, 2003.

[11] C. Cleaveland, "Building Application Generators", *IEEE Software*, July, 1988.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and Mixins". *ACM POPL*, San Diego, California, 1998, 171-183.

[13] G.Y. Guo, "A Software Architecture Reconstruction Method", *WICSA 1999*, 15-33.

[14] K. Kang, et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon Univ., Pittsburgh, PA, Nov. 1990.

[15] G. Kiczales, et al. "An overview of AspectJ". *ECOOP 2001*, Budapest, Hungary, 18-22.

[16] A. von Mayrhauser and A.M. Vans, "Program Understanding: Models and Experiments", Advances in Computers, Vol. 40, Academic Press, 1995, 1-38.

[17] J. Neighbors, "Software construction using components". Ph. D. Thesis, TR-160, University of California, Irvine, 1980.

[18] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database System", *ACM SIGMOD 1979*, 23-34.

[19] M. Stickel, et al., "Deductive Composition of Astronomical Software from Subroutine Libraries", In *Automated Deduction*, A. Bundy, ed., Springer-Verlag Lecture Notes in Computer Science, Vol. 814.

[20] R.E.K. Stirewalt and L.K. Dillon, "A Component-Based Approach to Building Formal Analysis Tools", *International Conference on Software Engineering*, 2001, 57-70.

[21] A.M. Vans, A. von Mayrhauser and G. Somlo, "Program Understanding Behavior During Corrective Maintenance of Large-Scale Software", *Int. J. Human-Computer Studies* (1999), #51, 31-70.

[22] T. Veldhuizen, "Using C++ Template Metaprograms", *C++ Report*, vol. 7, no. 4, May 1995, 36-43.