To appear. Invited presentation at the 10th International Conference on Algebraic Methodology And Software Technology AMAST'2004, Stirling, Scotland, July 12-16.

A Science of Software Design

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78746
batory@cs.utexas.edu

Underlying large-scale software design and program synthesis are simple and powerful algebraic models. In this paper, I review the elementary ideas upon which these algebras rest and argue that they define the basis for a science of software design.

1 Introduction

I have worked in the areas of program generation, software product-lines, domain specific languages, and component-based architectures for over twenty years. The emphasis of my research has been on large-scale program synthesis and design automation. The importance of these topics is intuitive: higher productivity, improved software quality, lower maintenance costs, and reduced time-to-market can be achieved through automation.

Twenty years has given me a unique perspective on software design and software modularity. My work has revealed that large scale software design and program synthesis is governed by simple and powerful algebraic models. In this paper, I review the elementary ideas on which these algebras rest. To place this contribution in context, a fundamental problem in software engineering is the abject lack of a science for software design. I will argue that these algebraic models can define the basis for such a science.

I firmly believe that future courses in software design will be partly taught using domain-specific algebras, where a program's design is represented by a composition of operators, and design optimization is achieved through algebraic rewrites of these compositions. This belief is consistent with the goal of AMAST. However, I suspect that *how* I use algebras and their relative informality to achieve design automation is unconventional to the AMAST community. As a background for my presentation, I begin with a brief report on the 2003 Science of Design Workshop.

2 NSF's Science of Design Workshop

In October 2003, I attended a *National Science Foundation (NSF)* workshop in Airlie, Virginia on the "Science of Design" [11]. The goal of the workshop was to determine the meaning of the term "Science of Design". NSF planned to start a program with this title and an objective was to determine lines of research to fund. There were 60 attendees from the U.S., Canada, and Europe. Most were from the practical side of software engineering; a few attendees represented the area of formal methods. I was interested in the workshop to see if others shared my opinions and experiences in software design, but more generally, I wanted to see what a cross-section of today's Software Engineering community believed would be the "Science of Design". In the following, I review a few key positions that I found particularly interesting.

Richard Gabriel is a Distinguished Engineer at Sun Microsystems and one of the architects of Common Lisp. He described his degree in creative writing — in particular, poetry — and demonstrated that it was far more rigorous in terms of course work than a comparable degree in Software Engineering (of which software design was but a small part). He advocated that students should be awarded degrees in "Fine Arts" for software design. I was astonished: I did not expect to hear such a presentation at a *Science* of Design workshop. Nevertheless, Gabriel reinforced the common perception that software design is indeed an art, and a poorly understood art at that.

Carliss Balwin is a Professor at the Harvard Business School. She argued that software design is an instance of a much larger paradigm of product design. She observed that the processes by which one designs a table, or a chair, or an auditorium, are fundamentally similar to that of designing software. Consequently, software design has firm roots in economic processes and formalisms. Once again, I was not expecting to hear such a presentation at a *Science* of Design workshop. And again, I agreed with her arguments that software design can be viewed as an application of economics.

Did the workshop bring forth the view of is software design as a *science*? I did not see much support for this position. Attendees were certainly using science and scientific methods in their investigations. But I found little consensus, let alone support, for software design as a science. The most memorable summary I heard at the workshop was given by Fred Brookes, the 1999 ACM Turing Award recipient. He summarized the conclusions of his working group as "We don't know what we're doing, and we don't know what we've done!".

The results of the workshop were clear: if there is to be a science of software design, it is a very long way off. In fact, it was questionable to consider software design a "science". Although I do not recall hearing this question posed, it seemed reasonable to ask if design is engineering.[1] For example, when bridges are designed, there is indeed an element of artistry in their creation. But there is also an underlying science called physics that is used to determine if the bridge meets its specifications. So if software design is engineering, then what is the science that underlies software design? Again, we are back to square one.

After many hours of thought, I realized that the positions of Gabriel and Baldwin were consistent with my own. Software design is an art as Gabriel argued, *but not always*. Consider the following: designing the first automobiles was an art — it had never been done before, and required lots of trial and error. Similarly, designing the first computer or designing the first compiler were also works of art. There were no assembly lines

1. Thanks to Dewayne Perry for this observation.

for creating these products and no automation. What made them possible was craftsmanship and supreme creativity. Over time, however, people began building variants of these designs. In doing so, they learned answers to the important questions of *how* to design these products, *what* to design, and most importantly, *why* to do it in a particular way. Decision making moved from subjective justifications to quantitative reasoning. I am sure you have heard the phrase "we've done this so often, we've gotten it down to a science". Well, that *is* the beginnings of a science.

A distinction that is useful for this paper is the following: given a specification of a program and a set of organized knowledge and techniques, if "magic" (a.k.a. inspiration, creativity) is needed to translate the specification into a program's design, then this process is an *art* or an *inexact science*. However, if it is purely a mechanical process by which a specification is translated into a design of an efficient program, then this process follows an *exact* or *deterministic science*.

Creating one-of-a-kind designs will always be an art and will never be the result of an exact or deterministic science, simply because "magic" is needed. Interestingly, the focus of today's software design methodologies is largely on creating one-of-a-kind products. The objective is to push the envelope on a program or component's capabilities, relying on the creativity and craftsmanship of its creators — and not automation. In contrast, I believe that an exact science for software design lies in the mechanization and codification of well-understood processes, domain-expertise, and design history. We have vast experiences building particular kinds of programs, we know the *how*, the *what*, and the *why* of their construction. We want to automate this process so that there is no magic, no drudgery, and no mistakes. The objective of this approach is *also* to push the envelope on a program or component's capability but *with emphasis on design automation*. That is, we want to achieve the same goals of conventional software development, *but from a design automation viewpoint.*

The mindset to achieve higher levels of automation is unconventional. It begins with a declarative specification of a program. This specification is translated into a design of an efficient program, and then this design is translated to an executable. To do all this requires significant technological advances. First, how can declarative specifications of programs be simplified so that they can be written by programmers with, say, a high-school education? This requires advances in *domain-specific languages*. Second, how can we map a declarative specification to an efficient design? This is the difficult problem of *automatic programming*; all but the most pioneering researchers abandoned this problem in the early 1980's as the techniques that were available at that time did not scale [1]. And finally, how do we translate a program's design to an efficient executable automatically? This is *generative programming* [9]. Simultaneous advances on all three fronts are needed to realize significant benefits in automation.

To do all this seems impossible, yet an example of this futuristic paradigm was realized *over 25 years ago,* around the time that others were giving up on automatic programming. The work was in a significant domain, and the result had a revolutionary impact on industry. The result: *relational query optimization (RQO)* [12].

Here's how RQO works: an SQL query is translated by a parser into an inefficient relational algebra expression. A query optimizer optimizes the expression to produce a semantically equivalent expression with better performance characteristics. A code generator translates the optimized expression into an efficient executable. SQL is a prototypical declarative domain-specific language; the code generators were early examples of generative programming, and the optimizer was the key to a practical solution to automatic programing.

In retrospect, relational database researchers were successful because they automated the development of query evaluation programs. These programs were hard to write, harder to optimize, and even harder to maintain. The insight these researchers had was to create an *exact* or *deterministic science* to specify and optimize query evaluation programs. In particular, they identified the fundamental operators that comprised the domain of query evaluation programs, namely relational algebra. They represented particular programs in this domain by expressions (i.e., compositions of relational operators). And they used algebraic identities to rewrite, and thus optimize, relational algebra expressions.

RQO is clearly an interesting paradigm for automated software development [5]. I cannot recall others ever proposing to generalize the RQO paradigm to other domains. The reason is clear: the generalization is not obvious. It is possible, and in the next sections, I show how.

3 Feature Oriented Programming

Feature Oriented Programming (FOP) originated from work on product-line architectures. The goal is to declaratively specify a program by the features that it is to have, where a feature is some characteristic that can be shared by programs in a domain. So program **P1** might have features **X**, **Y**, and **Z**, while program **P2** has features **X**, **Q**, and **R**. Features are useful because they align with requirements: customers know their requirements and can see how features satisfy requirements.

Interestingly, feature specifications of products are quite common. (It just isn't common for software). The Dell web site, for example, has numerous web pages where customers can declaratively specify the features they want on their PCs. A Dell web page is a declarative DSL; clicking the check boxes and selecting items from pull-down menus is the way declarative specs are written. By sending Dell a check for the computed amount, that customized PC will be delivered in days. Similarly, ordering a customized meal at a restaurant involves choosing items from a menu; this too is a familiar form of declarative specifications. Neither customizing PCs or ordering customized meals requires an advanced technical degree. We want the same for software.

GenVoca is a simple and powerful algebraic model of FOP. GenVoca is based on the idea of *step-wise refinement*, which is an ancient methodology for building software by progressively adding details [14]. The novelty of GenVoca is that it scales the concept of refinement. That is, instead of composing hundreds or thousands microscopic program rewrites called *refinements*, GenVoca scales refinements so that they each encapsulate an individual feature. A complete program is synthesized by composing a few feature refinements. (Warning: I am using the term "refinement" in its common *object-oriented (OO)* usage, namely to elaborate or extend. In contrast, "refinement" has a different meaning in algebraic specifications — it means to elaborate but *not* extend a program's functionality. "Extension" is a more appropriate term. Henceforth, I use the term "extension", but beware that papers on FOP use the term "refinement" instead).

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       // an application with feature f
h       // an application with feature h
```

The remaining operators are *functions* which represent program extensions:

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

The design of an application is a named expression called an *equation*:

```
app1 = i(f)           // application with features i and f
app2 = j(h)           // application with features j and h
app3 = i(j(h))        // application with features i,j,h
```

The family of programs that can be created from a model is its *product-line*. To simplify notation, we henceforth write **i(j(h))** as **i•j•h**, where • denotes function composition.

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. For example, suppose function $\mathbf{k}_1$ adds feature **k** with implementation #1 to its input program, while function $\mathbf{k}_2$ adds feature **k** with implementation #2. 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 straightforward. Figure 1 depicts a program **P** that is a package of four classes (**class1**—**class4**). These classes are synthesized by composing features **X**, **Y**, and **Z**. **X** encapsulates a fragment of **class1**—**class3**, which is shown in a solid color. **Y** extends **class1**—**class3** and introduces

class4, which is shown in horizontal stripes. **Z** extends all four classes, and is shown in checker-board. Thus features encapsulate fragments of classes. Composing features yields packages of fully-formed classes.

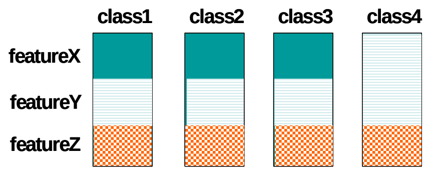


Fig. 1. Package **P = Z•Y•X**

My colleagues and I have had considerable success using GenVoca for creating product-lines for database systems [7], network protocols [7], data structures [6], avionics [2], extensible Java and Scheme compilers [3][10], and program verification tools [13]. The next section briefly explains how code synthesis is performed.

3.1 Implementation Details

Extension and composition are very general concepts that can be implemented in many different ways. The core approach relies on inheritance to express method and class extensions. Figure 2a shows method **A()** whose body sequentially executes statements **x**, **y**, and **z**. Figure 2b declares an extension of this method to be **Super.A()** followed by statement **w**. **Super.A()** says invoke the method's original definition. The composite method is Figure 2c; it was produced by substitution (a.k.a. macro-expansion): that is, **Super.A()** was replaced with the original body of **A()**.

(a)
```
void A() {
   x; y; z;
}
```

(b)
```
void A() {
   Super.A(); w;
}
```

(c)
```
void A() {
   x; y; z; w;
}
```

Fig. 2. Method Definition and Extension

Class extensions are similarly familiar. Figure 3a shows a class **P** that has three members: methods **A()**, **B()**, and data member **C**. Figure 3b shows an extension of **P**, which encapsulates extensions to methods **A()** and **B()** and adds a new data member **D**. The composition of the base class and extension is Figure 3c: composite methods **A()** and **B()** are present, plus the remaining members of the base and extension.

```
class P {
  void A(){ x;y;z; }
  void B(){ r;t; }
  int C;
}
```
(a)

```
refines class P {
   void A(){ Super.A();w; }
   void B(){ q;Super.B(); }
   String D;
}
```
(b)

```
class P {
  void A(){x;y;z;w;}
  void B(){q;r;t;}
  int C;
  String D;
}
```
(c)

Fig. 3. Method Definition and Extension

One way to implement the above is to use subclassing: that is, make Figure 3b a subclass of Figure 3a, where the semantics of the subclass equals that of Figure 3c. Another way is to use substitution (in-place modification) as we have illustrated. There are many other ways to realize these ideas with or without inheritance.

4 AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) is the successor to GenVoca [3]. It embodies ideas that have revolutionized my thinking on program synthesis. In particular, AHEAD shows how step-wise development scales 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 of AHEAD.

4.1 Multiple Program Representations

Generating code for individual programs is not sufficient. 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 a system's design, such as process models, UML models, makefiles, or formal specifications.

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, while another an HTML document (which might be the program's design document). There are class file or binary representations of a program, makefile representations, performance models, and so on. A program should encapsulate all of its artifacts or projections.

In general, suppose program $\mathbf{P}$ encapsulates artifacts $\mathbf{A}_p$, $\mathbf{B}_p$, and $\mathbf{C}_p$, where the meaning of these artifacts is uninterpreted. I express this relationship algebraically as:

$$\mathbf{P} = \{ \mathbf{A}_p, \mathbf{B}_p, \mathbf{C}_p \}$$

where set notation denotes encapsulation. Members of a set are called *units*.

4.2 Generalize Extensions

Adding a new feature to a program may change any or all of its representations. For example, if a new feature $\mathbf{F}$ is added to program $\mathbf{P}$, one would expect changes in $\mathbf{P}$'s code (to implement $\mathbf{F}$), documentation (to document $\mathbf{F}$), makefiles (to build $\mathbf{F}$), formal properties (to characterize $\mathbf{F}$), performance models (to profile $\mathbf{F}$), and so on.

In general, suppose feature $\mathbf{F}$ changes artifacts $\mathbf{A}$ and $\mathbf{B}$ (where $\mathbf{A}_f$ and $\mathbf{B}_f$ denote the specifications of these changes) and adds new artifact $\mathbf{D}_f$. I say $\mathbf{F}$ encapsulates $\mathbf{A}_f$, $\mathbf{B}_f$, and $\mathbf{D}_f$, and write this relationship algebraically as:

$$\mathbf{F} = \{ \mathbf{A}_f, \mathbf{B}_f, \mathbf{D}_f \}$$

4.3 Generalize Composition

Given **P** and **F**, how is **F•P** computed? The answer: composition is governed by rules of inheritance. Namely, all units of the parent (inner or right-hand-side) feature are inherited by the child (outer or left-hand-side) feature. Further, units with the same name (ignoring subscripts) are composed pairwise with the parent term as the inner term:

$$\begin{aligned} \mathbf{F}\bullet\mathbf{P} \quad &= \{\ \mathbf{A}_f,\ \mathbf{B}_f,\ \mathbf{D}_f\ \} \bullet \{\ \mathbf{A}_p,\ \mathbf{B}_p,\ \mathbf{C}_p\ \} \\ &= \{\ \mathbf{A}_f \bullet \mathbf{A}_p,\ \mathbf{B}_f \bullet \mathbf{B}_p,\ \mathbf{C}_p,\ \mathbf{D}_f\ \} \qquad \textbf{(1)} \end{aligned}$$

Stated another way, **F•P** is computed by composing corresponding artifacts and the correspondence is made by name. Thus, the **A** artifact of **F•P** is produced by $\mathbf{A}_f\bullet\mathbf{A}_p$ — the original artifact $\mathbf{A}_p$ extended by $\mathbf{A}_f$. Similarly, the **B** artifact of **F•P** is $\mathbf{B}_f\bullet\mathbf{B}_p$ — the original artifact $\mathbf{B}_p$ extended by $\mathbf{B}_f$. Artifacts **C** and **D** of **F•P** correspond to their original definitions. **(1)** defines the *Law of Composition*: it tells us how *composition distributes over encapsulation*.

Readers may recognize Figure 3 to be a particular example of this law. **P** is the base class of Figure 3a, encapsulating members **A**, **B**, and **C**. **F** is the class extension of Figure 3b, encapsulating members **A**, **B**, and **D**. The composition **F•P** — an illustration of **(1)** — is Figure 3c. More on this in the next section.

You will see shortly that the Law of Composition applies at all levels of abstraction and can be made to apply to all artifacts. Figure 4 is an example of the latter. Figure 4a is a grammar of a language that sums integers. Figure 4b shows a grammar extension that adds the minus operation. In particular, a new token **MINUS** is added to the grammar and the **Operator** production is extended with the **MINUS** rule. (The phrase **Super.Operator** says substitute the right-hand-side of the original **Operator** production). Figure 4b shows the composite grammar. Each token and production corresponds to individual terms in the Law of Composition.

```
// INTEGER is predefined

"+"  PLUS

Expr
    : Val
    | Val Operator Expr
    ;

Val
    : INTEGER
    ;

Operator
    : PLUS
    ;
```
(a) constant

```
"-" MINUS

Operator
    : Super.Operator
    | MINUS
    ;
```
(b) function

```
"-"            MINUS
"+"            PLUS

Expr
    : Val
    | Val Operator Expr
    ;

Operator
    : PLUS
    | MINUS
    ;

Val
    : INTEGER
    ;
```
(c) composition

Fig. 4. Grammars, Extensions, and Composition

4.4 Generalize Modularity

A *module* is a containment hierarchy of related artifacts. Figure 5a 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.

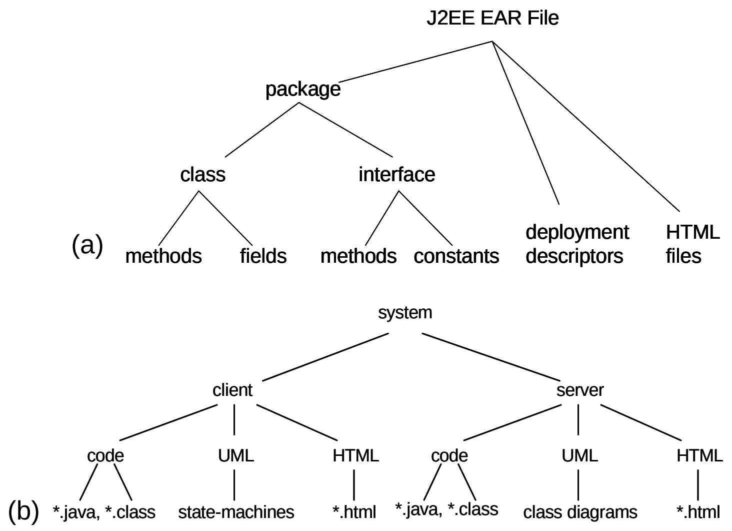


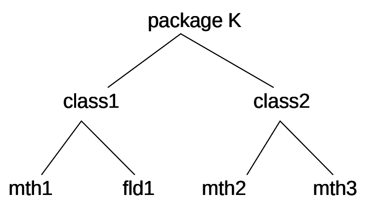
Fig. 5. Modules are Containment Hierarchies

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 5b 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 simple algebraic representations as nested sets of constants and functions. Figure 6a shows package **`K`** to encapsulate **`class1`** and **`class2`**, **`class1`** encapsulates method **`mth1`** and field **`fld1`**. **`class2`** encapsulates **`mth2`** and **`mth3`**. The corresponding set notation is shown in Figure 6b.

4.5 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 the representations of a base program. An AHEAD function is a hierarchy of extensions — that is, it is a containment hierarchy that can add new artifacts (e.g., new Java and HTML files), and can also refine/extend existing artifacts. When features are composed, corresponding program representations are composed. If the representations of



(a) **graphical**

K = { class1, class2 }

class1 = { mth1, fld1 }

class2 = { mth2, mth3 }

(b) **algebraic**

Fig. 6. Modules and Nested Sets

each feature are consistent, then their composition is consistent. Thus consistent representations of programs can be synthesized though composition; this is exactly what is needed.

4.6 Implementation Details

We implement module hierarchies as directory hierarchies. Figure 7a shows our algebraic representation of a module, and Figure 7b shows its directory representation.

```
(a) A = { Code, R.drc, Htm }

    Code = { X.java, Y.java }

    Htm = { W.htm, Z.htm }
```

(b)

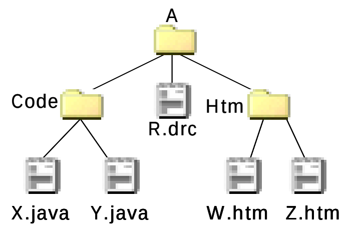


Fig. 7. Corresponding Algebraic and Directory Representations

Feature composition is directory composition. That is, when features are composed, their corresponding directories are folded together to produce a directory whose structure is isomorphic to the feature directories that were composed. For example, the **`X.java`** file of **`C = B•A`** in Figure 8 is produced by composing the corresponding **`X.java`** files of **`B`** and **`A`**.

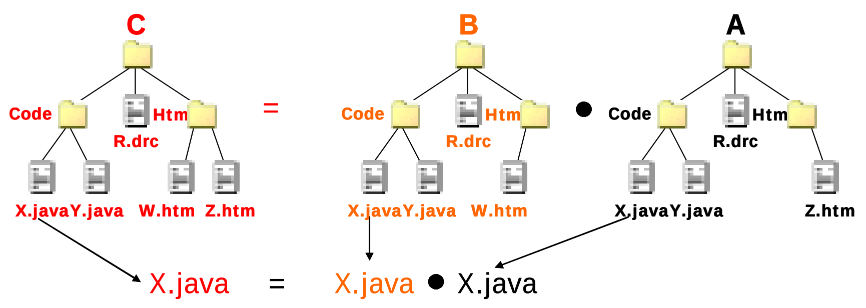


Fig. 8. Composition of Feature Directories

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

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

The result is a nested set of expressions. Each expression tells us how to synthesize an artifact of the target system. 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 9 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. This expression 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 with respect to the original declarative specification. AHEAD is a generalization of the Relational Query Optimization paradigm.

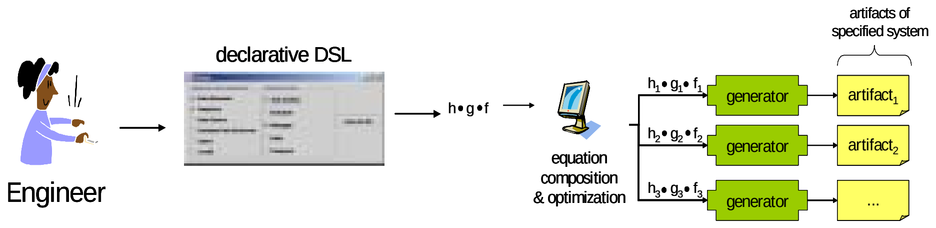

artifacts of specified system
declarative DSL
Engineer
h•g•f
equation composition & optimization
$h_1\bullet g_1\bullet f_1$
$h_2\bullet g_2\bullet f_2$
$h_3\bullet g_3\bullet f_3$
generator
generator
generator
artifact_1
artifact_2
...

Fig. 9. Program Synthesis Paradigm of AHEAD

A common question that is asked is: can you realistically design features so that they can be composed? Absolutely. It's easy — this is what software product-lines are all about. Features or components are designed to be composable and compatible. Composability and compatibility are properties that don't happen by magic or by accident; they are premeditated. Some of you may recall the old chip catalogs of the early 1970s, where all the chips in the catalog were designed to be compatible — they worked off of the same voltage, impedance, etc. Chips built by another manufacturer often were not compatible. A more familiar example today are disks for PCs. There are all sorts of disk manufacturers now that have an incredible line of products. These disks are compatible because they meet SCSI or IDE standards. (Recall that prior to plug-and-play standards, adding a disk to a PC required a high-paid technician to do the installation). The same ideas apply to software.

5 Cultural Enrichment

It is beyond the scope of this paper to show a detailed example of these ideas in action. For those interested, please consult [4]. In this section, I explain a simple result that illustrates AHEAD algebras and an elementary optimization. Then I discuss the breadth of the AHEAD framework.

5.1 A Simple Result

Have you ever wondered what an algebraic justification of object-oriented subclassing (e.g., inheritance) would be and why inheritance is fundamental to software designs? Here's an answer: when a program is synthesized, an expression is generated for every class of that program. Suppose the program has classes **`A`**, **`B`**, and **`C`** with the following expressions:

```
A = Z • Y • X • W
B = Q • Y • X • W
C = E • Y • X • W
```

Observe that all three classes share a common sub-expression **`Y•X•W`**. Instead of redundantly evaluating this expression, one can eliminate common sub-expressions by *factoring*:

```
F = Y •X •W
A = Z •F
B = Q •F
C = E •F
```

Whenever a new equation **`F`** is created, a new class **`F`** is generated. The relationship between classes **`F`** and **`A`**, **`B`**, **`C`** is indicated in the revised expressions: the expressions for **`A`**, **`B`**, **`C`** reference and thus *extend* **`F`**. Code generators materialize **`F`** as a class with **`A`**, **`B`**, and **`C`** as subclasses. That is, **`F`** contains the data members and methods that are common to its subclasses. *Just as common sub-expression elimination is fundamental to algebra, inheritance hierarchies are fundamental to object-oriented designs, because they are manifestations of the same concept*. Interestingly, the process of promoting common methods and data members of subclasses into a superclass is called *refactoring* in OO parlance, whereas in algebra it is called *factoring*. Not only are the concepts identical, the names are almost identical too.

5.2 Even More Generality

I have concentrated so far on *domain-specific* operators — constants and functions — whose compositions define programs within a target domain. But there are many operators in the AHEAD universe that are not domain-specific. I illustrate some with an example.

In the current implementation of AHEAD, all code is written in Java that has been extended with refinement constructs (e.g., "**`refines`**" and "**`Super`**"). This language is called Jak (short for "Jakarta"). To compile these classes, the Jak files are translated to their Java file equivalents using the **`jak2java`** tool. Next, the Java files are translated to class files using the **`javac`** tool. These are *derivation steps* that can be expressed algebraically: Let **`P`** be a program that is synthesized from some AHEAD equation. **`P`** encapsulates a set of Jak files. Let **`P'`** define the set that additionally contains the corresponding Java and class files. The relationship of **`P`** and **`P'`** is expressed as:

```
P' = javac( jak2java( P ) )                                    (2)
```

That is, the **`jak2java`** tool is an operator that elaborates or extends containment hierarchies by translating every Jak file to its corresponding Java file. Similarly, the **`javac`** tool is an operator that elaborates containment hierarchies by translating every Java file to its corresponding class file(s).

There are many other operators on containment hierarchies, such as **`javadoc`** (that derives HTML-documentation files from Java files), **`javacc`** (that derives Java files that implement a parser from .jj files), and **`jar`** (a Java archive utility). In short, common tools that programmers use today are operators that derive and thus add new representations to containment hierarchies. Readers will recognize equation **`(2)`** to be an algebraic specification of a makefile. It is *much* easier to write and understand **`(2)`** than its corresponding **`ant`** declaration in XML. Thus, it should be possible to generate low-level makefiles from equational specifications like **`(2)`**. Further, if the makefile expression is particularly complicated, it may be possible also to *optimize* the expression automatically (e.g., perform **`javac`** operator before **`javadoc`**) prior to makefile generation. Doing so would further relieve programmers of the burden of manually writing and optimizing these files.

Software design involves many activities that involves many representations of a program (e.g., analysis, implementation, description). Given the ability to compose and derive, you can do just about anything. AHEAD unifies these activities in an algebraic setting. Not only will this simply program specifications, it will also make specifications more amenable to automatic optimization.

6 Why Does AHEAD Work?

I'm sure that some of you, upon reading this paper, will wonder what the big deal is; the ideas are straight from an introductory computer science course (a.k.a. "CS 101"). You are right about the simplicity, but you may have forgotten the state (and abject lack) of science in software design (e.g., Section 2). As computer scientists we understand CS 101 concepts, but I claim that we do *not* know how they are applied to *software design*. Software engineers do *not* think about software designs in terms of algebras or anything (as far as I can tell) dealing with mathematics. They certainly do not think of design in terms of automation. Their reward structure for software development is also screwed up: complexity is appreciated; simplicity and elegance are dismissed. Not only is this bad science, this is bad engineering.

My students and I have synthesized the AHEAD tool suite, which is in excess of 250K Java LOC, from elementary equational specifications. We know of no technical limit on the size of system that can be synthesized. The obvious questions are: why can AHEAD be this simple? What are we giving up? And why does it work as well as it does?

There are many answers; here is one. I have seen the following idea proposed by different people in three consecutive decades, starting in late 1970s. Suppose a pair of independently written packages **Q** and **R** are to be composed. Composition is accomplished in the following way: corresponding classes in **Q** and **R** are identified. That is, class **X** in **Q** corresponds to class **Y** in **R**, and so on for every relevant class. Then corresponding methods are paired. That is, method **M** in class **X** of **Q** corresponds to method **N** in class **Y** of **R**, and so on for all relevant methods. And finally, corresponding parameters are matched: parameter 2 of method **M** in class **X** of **Q** corresponds to parameter 3 of method **N** in class **Y** of **R**, etc. Given these relationships, tools can be built to compose **Q** and **R**, and examples can show that indeed a package with the functionalities of **Q** and **R** can be created.

This approach has many problems. First, it does not scale in general. What happens if **Q** and **R** both have 1000 classes, 10,000 methods, and 20,000 parameters? Who can define (or have the patience to define) the required correspondences? How does one validate these correspondences? Note the hidden complexity of this approach: every concept (class, method, parameter) has *two* names: the name used in package **Q** and that used in **R**. This means that programmers have to remember *twice* the amount of information than they need to. If *k* additional packages are to be composed, then programmers must know *k* different names for the *same* concept. The concept is the essence of this problem; the different arbitrary names add *accidental complexity* [8]. In general, programmers have a hard time remembering all these accidental details.

Second, it does not work in general. Just because the names in packages can be aligned syntactically does not imply semantic alignment. That is, there is no guarantee that corresponding methods agree on their semantics. If there is disagreement, this approach to composition and software synthesis fails. The only recourse is to define an adaptor that performs the semantic translation — if in fact such an adaptor can be written.

In my experience, software reuse and software composition succeeds because of a premeditated design. Stated another way, components are composable *only* if they were *designed* with the other in mind. *This axiom is crucial to scalability.* It is very easy (and wrong) to assume that just because a small example works where this axiom does not hold, more complicated examples will work just as well.

AHEAD's contribution is basic: it shows how simple ideas and common software development tools can be unified and elegantly captured as an algebra that expresses the essential tasks of software design and scalable software generation as mathematics. I reject axioms that accept accidental complexity as fundamental. As a consequence, AHEAD provides a clean solution to a core design problem where class, method, and parameter names are standardized, and so too are their semantics. This enables us to build simple tools and attack problems of automated design on scale. Our work relies on a common engineering technique: arbitrary and random complexity are eliminated *by design*. AHEAD does indeed have correspondence mappings; they are implicit, and hence easy to write and easy to understand.[2]

The following quote from Fred Brooke's 1987 "No Silver Bullet" paper [8] helps to put the above arguments into context:

> The complexity of software is an essential property. Descriptions of software that abstract away its complexity often abstract away its essence…[3] Software people are not alone in facing complexity. Physicists deal with terribly complex objects even at the "fundamental" particle level. The physicist labors on in a firm faith that there are unifying principles to be found … Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary.
>
> No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced without rhyme or reason, because they were designed by different people, rather than by God.

Not quite. Just like physicists, I too believe there is an underlying simplicity in software. Programmers are geniuses in making simple things look complicated. If software were truly complicated, people could not program and architects could not design. There *must* be an underlying simplicity. The challenge is not making things complicated, but revealing their essential simplicity. That is what AHEAD does and this is why it scales.

Further, it does not take God to eliminate complexity. All it requires is common sense and the use of common engineering techniques. This has worked for many non-software domains; I and others are showing that it can work for software as well.

7 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 are modules? And how are modules composed to build larger modules? Unfortunately, the structure of software is not yet well-understood. Software design, which is the process to define the structure of an application, is an art. As long as it remains an art, our abilities to automate software design and make software development a true engineering discipline are limited.

In short, software design is in desperate need for a science of design. Such a science, I have argued, must be intimately related to automated design. That is, a science of software design will not arise from having virtuoso engineers use their creativity and

2. Our belief is that once the core problems are understood, one can begin to relax assumptions within the AHEAD framework itself to address progressively broader design problems.

3. The exception, of course, is modularity; hiding the details is what modularity is all about.

craftsmanship to create one-of-a-kind products. A science of design must arise from domains where the software development process is mature or reasonably well-understood and where developers can mechanize the process of creating successive variants of programs. Because today's models of software design are not aimed at automation, progress towards a science is understandably lacking.

I believe a science of software design is indeed possible, and have argued that the seminal work on Relational Query Optimization (RQO) is a powerful paradigm that can be emulated in other domains. RQO showed how declarative specifications can be translated into efficient query evaluation programs automatically. The core reason why RQO was successful is because it expressed the design of query evaluation programs algebraically. I presented FOP as a generalization of the RQO paradigm. I believe FOP could be a basis for a science of software design for many reasons:

- it raises the level of modularity from "code" to "design-related-artifacts",
- program design and optimization are expressed algebraically (and thus are ideal for automatic manipulation),
- it allows us to reason about applications in terms of their features (as architects do), and
- it is based on a few simple ideas whose applicability shows no apparent bounds.

Historically, science has progressed by leaps of intuition followed by many years of community debate before the real contributions of a work are understood. The science of software design will be no different; such a science is indeed years away. AHEAD has the right "look and feel" for particular aspects of software design, but that is a far cry from having the software development community accept and appreciate its possibilities. FOP does seem to be a step in the right directions of simplicity, mathematical elegance, and support for automation. These criteria, more than anything else, are the metrics by which advances in software design should be measured.

Acknowledgements. I gratefully acknowledge help from Vicki Almstrum, Dewayne Perry, Chetan Kapoor, Chris Lengauer, and Carron Shankland on improving earlier drafts of this paper.

8 References

[1] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering*, November 1985.

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

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

[4] D. Batory, J. Liu, J.N. Sarvela, "Refinements and Multi-Dimensional Separation of Concerns", *ACM SIGSOFT 2003 (ESEC/FSE2003)*.

[5] D. Batory, "The Road to Utopia: A Future for Generative Programming", Dagstuhl on Domain-Specific Program Generation, Springer *LNCS* #3016, 1-17.

[6] D. Batory, et al., “Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Trans. Software Engineering*, May 2000, 441-452.

[7] D. Batory and S. O'Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. *ACM TOSEM*, 1(4):355-398, October 1992.

[8] F.P. Brookes, “No Silver bullet: Essence and Accidents of Software Engineering”, *IEEE Computer*, April 1987, pp. 10-19.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 2000.

[10]S. Krishnamurthi, “Linguistic Reuse”, Ph.D. Rice University, 2001.

[11]National Science Foundation, “Science of Design: Software-Intensive Systems”, Workshop Proceedings, October 2003.

[12]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.

[13]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.

[14]N. Wirth, “Program Development by Stepwise Refinement”, *CACM* Vol. 14, No. 4, April 1971, 221-227.