Revised September 8, 2004

# A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite (ATS)

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

## 1 Introduction

*Software Engineering (SE)* is in a perpetual crisis. Software products are increasing in complexity, the cost to develop and maintain systems is skyrocketing, and our ability to understand systems is decreasing. A basic goal of SE is to successfully manage and control complexity; the "crisis" indicates that SE technologies are failing to achieve this goal. There are many culprits. One surely is that today's software design and implementation techniques are simply too low-level, exposing far more detail than is necessary to make an application's design, its construction, ease of modification simple. Future software design technologies will need to do much better, and it should not be surprising that they will be quite different from those of today.

Looking to the future, SE paradigms will likely embrace the following ideas:

- *Generative Programming (GP)*
- *Domain-Specific Languages (DSLs)*
- *Automatic Programming (AP)*

GP is about automating software development. Eliminating the task of writing mundane and rote programs is a motherhood to improved programmer productivity and program quality. Program synthesizers will transform input specifications into target programs. These specifications will not be written in Java or C# — which are far too low-level — but rather in high-level notations called DSLs that are specific to a particular domain. DSL programs are known to be both easier to write and maintain than their low-level (e.g., Java) counterparts. Ideally, DSLs will be declarative, allowing their users to define *what* is needed and leave it up to the DSL compiler to automatically produce an efficient program that does the *how* part. But placing the burden of program synthesis on a DSL compiler should not be taken lightly. This is the problem of AP; it is a technical problem of great difficulty, as little progress has been made in the last 20 years to produce demonstrably efficient programs from declarative specs. Advancement on all three fronts (GP, DSLs, and AP) may be needed before the crisis in SE will noticeably diminish.

While it is wishful thinking that simultaneous advances on all three fronts is possible, it is worth noting that a spectacular example of this futuristic SE paradigm was realized over *20 years ago* — ironically around the time when others gave up on AP. And not only that, it had a fundamental impact on commercial applications. The example is relational query optimization. SQL is a prototypical DSL: it is a *declarative language* for retrieving data from tables. An SQL compiler translates an SQL statement into a relational algebra expression. A query optimizer accomplishes the goal of *automatic programming* by applying equational rewrite rules to automatically optimize relational algebra expressions. The task of translating an optimized expression into an efficient program is an example of *generative programming*.

Relational optimizers revolutionized databases: data retrieval programs that were hard to write, hard to optimize, and hard to maintain are now produced automatically. There is nothing special about data retrieval programs: all interesting programs are hard to write, optimize, and maintain. Thus if ever there was a "grand challenge" for SE, it is to replicate the success of relational query optimization in other domains.

AHEAD is a theory of *Feature Oriented Programming (FOP)* that shows how the concepts and framework of relational query optimization generalize to other domains. ATS is a suite of tools that implement the AHEAD theory.

### 1.1 Background

How would you describe a program that you've written to a prospective customer? You are unlikely to recite what DLLs you're using — because the customer would unlikely have any interest such details. Instead, you would take a more promising tact of explaining the *features* that your program offers its clients. This works because clients know their requirements and can see how features satisfy requirements.

Successful programs come in different flavors, e.g., entry-level through deluxe. The differences between these categories are the presence or absence of features (or more commonly, sets of features). Entry-level versions have a minimal feature set; deluxe advertises the most.

But if we describe programs by features or differentiate programs by features, why can't we build programs (or program families) from feature specifications? In fact, we can. This is the area of research called *product-lines*. The ability to add and remove features implies that features are modularized. We focus on a particular sub-topic of product-line research that deals with feature modularization. By making features first-class design and implementation entities, it is easier to add and remove features from applications. (In fact, this is a capability that most of us wish we had today — the ability to add and remove features from our programs. We don't have it now; the purpose of this paper is to explain how it can be done in a general case). It happens that feature modularity goes far beyond conventional notions of code modularity. This, among other things, makes it a very interesting topic.

*Feature oriented programming (FOP)* is the study of feature modularity and programming models that support feature modularity. A powerful form of FOP is based on a methodology called *step-wise refinement (SWR)*. SWR is both simple and ancient: it advocates that complex programs can be constructed from simple programs by incrementally adding details. When incremental units of change are features, FOP and SWR converge. This is the starting point of AHEAD and ATS. But what is a feature refinement? How is it represented? And how are refinements and their compositions modeled?

## 1.2 A Clue

Consider any Java class **C**. A class member could be a data field or a method. Class **C** below has four members **m1**—**m4**.

```
class C {
   member m1;
   member m2;
   member m3;
   member m4;
}                                          (1)
```

Have you ever noticed that when **C** belongs to an inheritance hierarchy, the members of **C** could be distributed over **C**'s ancestors? For instance, one possibility is to have class **C1** encapsulate member **m1** and **C23** encapsulate members **m2** and **m3**:

```
class C1 { member m1; }

class C23 extends C1 {
   member m2;
   member m3;
}
class C4 extends C23 { member m4; }
class C extends C4 {}                      (2)
```

From a programmatic viewpoint, both definitions of **C**, namely **(1)** and **(2)**, are equivalent. In fact, we could further decompose **C23** to be:

```
class C2 extends C1 { member m2; }
class C3 extends C2 { member m3; }
class C23 extends C3 {}
```

and the definition of **C** would not change; it would still have members **m1**—**m4**. Moreover, there's nothing really special about the placement of member **m1** (or **m2** …) in this hierarchy. If **m1** is a method and it references other members, as long as these members are not defined lower in the inheritance chain, it can appear in any class of the chain.

If you have taken a course on algebra, you have seen these ideas before. Consider sets and the union operator. We can define the sets:

```
C1 = { m1 }
C2 = { m2 }
C3 = { m3 }
C4 = { m4 }

C23 = C2 ∪ C3
C = C1 ∪ C23 ∪ C4 = C1 ∪ C2 ∪ C3 ∪ C4
```

Union is a commutative operator, which means that the order in which the union of sets is taken doesn't matter. This is similar to, but not exactly the same as inheritance because as we saw, a method can be introduced only as long as members it references are not defined in lower classes.

Something a bit closer to inheritance are vectors and the vector operations of addition(**+**) and movement(⟶). Suppose we define vectors in 4-space:

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

You know about vector arithmetic; vector movement is the path that is followed when laying vectors end-to-end. Vector arithmetic is commutative; vector movement isn't:

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

C1 ⟶ C2 ⟶ C3 ⟶ C4 ≠ C4 ⟶ C3 ⟶ C2 ⟶ C1
```

Inheritance has the flavor of both vector arithmetic and vector movement.

When you think about an *operator* for inheritance, what you are really defining is an operator for class refinement. A *class refinement* can add new members and refine existing members of a class. (Java doesn't support field refinements, but method overrides or extensions is method refinement).

Here's an example. Suppose a program **P** has a single class **B** that initially contains a single data member **x**:

```
class B { int x; }                                    (3)
```

Suppose a refinement **R** of program **P** adds data member **y** and method **z** to class **B**. This class refinement is written as:

```
refines class B {
   int y;
   void z() {...}
}                                                     (4)
```

The composition of **R** with **P** defines a new program **N** with a single class, namely **B**, with the following three members:

```
class B {
  int x;
  int y;
  void z() {...}
}                                                     (5)
```

In effect, this composition is expressed by the following inheritance chain, called a *refinement chain*:

```
class B_P { int x; }

class B_R extends B_P {
  int y;
  void z() {...}
}

class B_N extends B_R {}                              (6)
```

where subscripts indicate the program or refinement from which that fragment of **B** is defined.

We can express these ideas algebraically in terms of "constants" and "functions". Program **P** is a constant — it defines a base artifact. A refinement is a function that maps programs, so **R** is a function. A composition is an expression. Thus, we can model **(5)** and **(6)** as the equation **N = R(P)**.

We can express our previous example about class **C** in this manner. Here is one way: let **C1** be a constant and **C2**, **C3**, and **C4** be refinements:

```
class C { member m1; }          // constant C1

refines class C { member m2; } // function C2

refines class C { member m3; } // function C3

refines class C { member m4; } // function C4
```

Now, class **C** of **(1)** can be synthesized from the expression **C4(C3(C2(C1)))**. Moreover, we see that **C23** has an obvious representation as a composite function or composite refinement:

```
C23 = C2 • C3
```

where • is the function composition operator. (Notationally **R(P)** is equivalent to **R•P**. Eventually we will only use the • notation to denote composition). The refinement of **C23** is:

```
refines class C {
   member m2;
   member m3;
}                                                     (7)
```

There remains loose ends to tie up before a bigger picture emerges. First, there's scalability. The effects of a refinement need not be limited to a single Java class. In fact, it is common for a "large-scale" refinement to encapsulate refinements of multiple classes as well as adding new classes. That is, such a refinement would augment existing classes of a program with new members and override existing members, but would also introduce new classes that could be subsequently refined.

Second, refinements have meaning when they encapsulate the implementation of a feature. Have you ever added a new feature to an existing program? You discover that you often have to extend a number of classes, as well as introduce new classes to a program. Well, that's a "large scale" refinement called a *feature refinement*.

Third, in product-line design, feature refinements are stereo-typical units of application design that can be composed with other such design elements to produce customized programs. A design for a product-line — called a *domain model* — is a set of constants and functions, where constants correspond to base programs and functions are feature refinements. These "units" — constants and functions — are effectively the legos of a domain that can be snapped together (i.e., composed functionally) to synthesize customized programs.

Fourth, recall that a key to the success of relational query optimizers is that they used equational representations of programs. That is, a data retrieval program was defined by a unique composition of relational algebra operators. To see the generalization of these ideas to other domains, a domain model is a set of operators ("constants" and "functions") whose compositions define the space of programs that can be synthesized within a domain. Given equational representations, there will always be algebraic identities among operators. These identities can be used to optimize equational representations of programs, just like relational query optimizers.

Finally, we have equated the terms "function" and "operator" in our discussions, but a mathematician would object as the ideas are not the same. An *operator* is a higher-order function, i.e., a function that maps functions. The calculus operators of differentiation and integration are examples. So when we talk about programs and program refinements, are we talking about functions or operators?

The answer is operators. Programs have command line arguments, hence they are functions. Program refinements are program-to-program mappings (i.e., function-to-function mappings), and hence they must be operators. Thus, domain models really are *algebras* — a set of operators that are specific to a particular domain of programs. Compositions of these operators represent different programs in the domain.

Even though we are dealing with operators, we will continue to refer to them as functions (which indeed they are). Now, let's consider a more precise way to express these ideas.

## 2 A Model of FOP

Salient ideas of FOP as expressed by two models: GenVoca and its successor AHEAD.

### 2.1 GenVoca

GenVoca is a design methodology for creating application families and architecturally-extensible software, i.e., software that is customizable via module additions and removals. It follows traditional step-wise refinement with one major difference: instead of composing thousands of microscopic program refinements (e.g., **x+1⇒inc(x)**) to yield admittedly small programs, GenVoca scales refinements so that each adds a feature to a program, and composing few refinements yields an entire program.

In GenVoca, programs are *constants* and refinements are *functions* that add features to programs. Consider the following constants that represent programs with different features:

```
f       // program with feature f
g       // program with feature g
```

A *refinement* is a function that takes a program as input and produces a refined or feature-augmented program as output:

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

A multi-featured application is an *equation* that is a named expression. Different equations define a family of applications, such as:

```
app1 = i(f)     // app1 has features i and f
app2 = j(g)     // app2 has features j and g
app3 = i(j(f))  // app3 has features i, j, f
```

Thus, the features of an application can be determined by inspecting its equation.

Note that a function represents both a feature *and* its implementation — 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 *equation optimization* to determine which implementation of **k** is best (e.g., provides the best performance)[1]. It is possible to automatically design software (i.e., produce an equation that optimizes some criteria) given a set of declarative constraints for a target application.

Although GenVoca constants and functions seem untyped, typing constraints do exist as design rules. *Design rules* are domain-specific constraints that capture syntactic and semantic constraints that govern legal compositions. It is common that the selection of a feature will disable or enable the selection of other features.

### 2.2 AHEAD

AHEAD, or *Algebraic Hierarchical Equations for Application Design*, embodies several key generalizations of GenVoca. First, a system has many representations besides source code, including UML documents, makefiles, BNF grammars, documents, performance models, etc. A model of FOP must deal with all these representations.

Second, each representation is written in its own *domain-specific language*. The code representation of a program may be represented in Java, a machine executable representation may be a class file, a makefile representation could be an Ant XML file, a performance model may be a set of Mathematica equations, and so on. An FOP model must support an open-ended spectrum of DSLs to express arbitrary system representations.

Third, when a system is refined by the addition of a new feature, any or all of the representations of a system may be updated. Thus, the concept of refinement applies not only source code representations, but all representations as well.

1. Different equations represent different programs and equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program.

Fourth, FOP models must deal with a very general notion of modularity: *a module is a containment hierarchy of related artifacts*. A class is a module (1-level hierarchy) that contains a set of data members and methods. A package or JAR file is a module (2-level hierarchy) that contains a set of classes. A J2EE EAR file is a module (3-level hierarchy) that contains sets of packages or JAR files, HTML files, and descriptor files. Going further, a client-server system is also a module (a multi-level hierarchy) that contains representations of both client and server programs.

Given the above, the GenVoca model generalizes. A "constant" is module that defines a containment hierarchy of related artifacts of different types written in their own DSLs. A "refinement" is function that maps containment hierarchies. Thus, whenever a refinement is applied to a system (i.e., an AHEAD constant), any or all of the representations in this module (containment hierarchy) will be updated and new artifacts introduced. Thus, as AHEAD refinements are applied, all of the representations of a generative program will remain consistent. This is exactly what is needed.

The notations of AHEAD extend those of GenVoca. A model is a set of features that are "constants" or "functions" called *units*. An alternative name for set is a *collective*. Thus, model **M** is a collective of units:

```
M = { a, b, c, d, ... }
```

Individual units may themselves be collectives, recursively:

```
a = { x, y, z }
z = { r, q }
...
```

The nesting of sets (collectives) models a containment hierarchy or module. Composition of units is defined by the laws of *inheritance*. That is, given units **X** and **Y**:

```
X = { a_x, b_x, c_x      }
Y = { a_y,      c_y, d_y }
```

Composition of **Y** and **X**, denoted **Y•X**, is formed by "aligning" the units of **X** and **Y** that have the same name (ignoring subscripts) and composing:

```
Y•X = { a_y•a_x, b_x, c_y•c_x, d_y }          (8)
```

Composition is recursive: if units represent collectives, their compositions are expanded according to **(8)**.

To see the connection with inheritance, consider the following inheritance hierarchy which is a class representation of **(8)**. Assume **a** and **c** are methods, where $\mathbf{a}_y$ and $\mathbf{c}_y$ refine (or override) their super-methods $\mathbf{a}_x$ and $\mathbf{c}_x$:

```
class X {
   member a_x;
   member b_x;
   member c_x;
}

class Y extends X {
   member a_y;
   member c_y;
   member d_y;
}

class Y•X extends Y {}
```

How the composition operator • is defined depends on the artifact type. • is polymorphic: it can be applied to all artifacts (i.e., all artifacts can be composed/refined) but what composition/refinement means is artifact type dependent (i.e., how makefile artifacts are refined will be analogous to but not the same as how code artifacts are refined).

AHEAD representations lead to simple tools and implementations. While there are many ways in which containment hierarchies can be realized, the simplest way is to map containment hierarchies to file system directories. Thus a feature might encapsulate many Java files, class files, HTML files, etc. Feature composition corresponds to directory composition.

We'll see examples of these ideas in the following sections.

## 3 A Simple Example

Consider a family of elementary post-fix calculators[2]. Calculators in this family are differentiated on (a) the arithmetic constants **BigInteger** or **BigDecimal** that can be specified and (b) the set operations that can be performed on them, which includes addition, division, and subtraction.

A model that describes this family is **C**:

```
C = { Base, BigI, BigD, Iadd, Idiv,
      Isub, Dadd, Ddivd, Ddivu, Dsub }
```

The lone constant in this model is **Base**, which defines an empty **calc** (short for "calculator") class (Figure 1a). The refinements **BigI** and **BigD** introduce a 3-level stack of **BigInteger** or **BigDecimal** constants, respectively (Figure 1b-c).[3] **BigI** and **BigD** are mutually exclusive as the stack variables introduced by both have the same name, but are of different types. Thus, calculators either work on **BigInteger** or **BigDecimal** numbers, but not both.

2. Modeled after Hewlett-Packard calculators.
3. A **BigInteger** is an unlimited precision integer; a **BigDecimal** is an unlimited-precision, signed decimal number.

```
class calc { }
```
(a) base/calc.jak

```
refines class calc {
   void divide() {
      e0 = e0.divide( e1 );
      e1 = e2;
   }
}
```
(d) Idiv/calc.jak

```
refines class calc {
   void add() {
      e0 = e0.add(e1);
      e1 = e2;
   }
}
```
(e) Iadd/calc.jak and Dadd/calc.jak

```
import java.math.BigDecimal;

refines class calc {
   void divide() {
      e0 = e0.divide( e1,
         BigDecimal.ROUND_DOWN );
      e1 = e2;
   }
}
```
(f) Ddivd/calc.jak

```
import java.math.BigInteger;

refines class calc {
   static BigInteger zero = BigInteger.ZERO;
   BigInteger e0 = zero, e1 = zero, e2 = zero;

   void enter( String val ) {
      e2 = e1;
      e1 = e0;
      e0 = new BigInteger(val);
   }

   void clear() {
      e0 = e1 = e2 = zero;
   }

   String top() { return e0.toString(); }
}
```
(b) BigI/calc.jak

```
import java.math.BigDecimal;

refines class calc {
   static BigDecimal zero = new BigDecimal("0");
   BigDecimal e0 = zero, e1 = zero, e2 = zero;

   void enter( String val ) {
      e2 = e1;
      e1 = e0;
      e0 = new BigDecimal(val);
   }

   void clear() {
      e0 = e1 = e2 = zero;
   }

   String top() {  return e0.toString(); }
}
```
(c) BigD/calc.jak

**Figure 1. Files from the C model**

The refinements **Iadd**, **Idiv**, and **Isub** respectively introduce the **BigInteger** addition, division, and subtraction methods to the **calc** class (Figure 1d-e). The refinements **Dadd**, **Ddivd**, **Ddivu**, and **Dsub** do the same for **BigDecimal** methods (Figure 1f-g). Note that there are two mutually exclusive **BigDecimal** division refinements: **Ddivd** and **Ddivu**. **Ddivd** rounds answers down, **Ddivu** rounds up.

As you may have already noticed, these files look like Java programs, but the language that we are using is not Java but an extended Java language called *Jak* (short for "Jakarta"). Jak files have ".jak" extensions, like Java files have Java extensions.

A calculator is defined by an equation. Here are just a few calculator definitions:

```
i1 = Iadd•BigI•Base
i2 = Isub•Iadd•BigI•Base
i3 = Idiv•Iadd•BigI•Base

d1 = Dadd•BigD•Base
d2 = Dsub•Dadd•BigD•Base
d3 = Ddivd•Dadd•BigD•Base
```

Calculator **i1** offers **BigInteger** addition. **i2** also supports subtraction. **i3** has **BigInteger** addition and division. **d1**—**d3** are the corresponding calculators for **BigDecimal** numbers using rounded-down division. The code generated for the **i3 calc** class is shown in Figure 2.

```
layer i3;

import java.math.BigInteger;

class calc {
   static BigInteger zero = BigInteger.ZERO;
   BigInteger e0 = zero, e1 = zero, e2 = zero;

   void add() {
      e0 = e0.add(e1);
      e1 = e2;
   }

   void clear() {
      e0 = e1 = e2 = zero;
   }

   void divide() {
      e0 = e0.divide( e1 );
      e1 = e2;
   }

   void enter( String val ) {
      e2 = e1;
      e1 = e0;
      e0 = new BigInteger(val);
   }

   String top() { return e0.toString(); }
}
```

**Figure 2. i3/calc.jak**

## Model Exercises

[1] What other calculator features could be added to **C**? What would be their Jak definitions? Look at the **BigInteger** and **BigDecimal** pages in J2SDK documentation for possibilities.

[2] Suppose the size of the stack were to be made variable. How would this be expressed as a refinement? What modifications of existing refinements would be needed?

[3] Modify model **C** so that **BigDecimal** round-up and round-down are features, which could parameterize operations like division.

[4] How would C be modified to permit the synthesis of a program that would invoke the calculator from the command-line? From a GUI?

## Tool Exercises

An AHEAD model **C** corresponds to a directory **C**, and each unit **U** in **C** corresponds to a subdirectory of **C**, namely **C/U**. The contents of a unit in our example is merely a **calc.jak** file. The AHEAD directory structure of **C** is:

```
C/Base/calc.jak     // see Figure 1a
C/BigI/calc.jak     // see Figure 1b
C/BigD/calc.jak     // see Figure 1c
C/Iadd/calc.jak     // see Figure 1d
C/Idiv/calc.jak     // see Figure 1e
C/Isub/calc.jak
C/Dadd/calc.jak     // see Figure 1d
C/Ddiv/calc.jak     // see Figure 1f
C/Dsub/calc.jak
```

Although we provide no **calc.jak** files for **Isub** and **Dsub**, they are easy to write. In fact, they are identical just like the **calc.jak** files for **Iadd** and **Dadd**.[4]

The **composer** tool is used to evaluate equations. **composer** has lots of optional parameters. For our tutorial, we need to reset one of these parameters. Create in the model directory a file called **composer.properties**. Its contents is a single line (which says when composing Jak files, use the **jampack** tool):

```
unit.file.jak : JamPackFileUnit
```

To compose files, run **composer** in the model directory. The order in which model units are listed on the **composer** command line are inside-to-outside order, and the name of the composition is given by the **target** option. Thus, to evaluate **i3** use:

```
> cd C
> composer --target=i3 Base BigI Idiv Iadd
```

The result of the composition is the directory **C/i3**, which contains a single file, **calc.jak**, shown in Figure 2. Note that the order in which units are listed on the **composer** command line is in reverse order of an equation — base first, outermost refinement last.

Validate your Model Exercise solutions by implementing them using AHEAD tools.

## 3.1 Translating to Java

The **jak2java** tool converts synthesized Jak files to their Java counterparts:

```
> cd i3
> jak2java *.jak
```

The above command-line translates all Jak files (in our case, there is only one file — **calc.jak**) to their Java equivalents. Of course, these generated files can be compiled in the usual way:

```
> javac *.java
```

## 3.2 Design Rules

New arithmetic operations could be added to **C** to greatly enlarge the family of calculators that can be synthesized. At the same time, it becomes increasingly clear that not all compositions are meaningful. In fact, it is quite easy to deliberately or unintentionally specify meaningless compositions, but composer is quite happy (typically) to generate code for these compositions. We need automated help to detect such situations.

*Design rules* are domain-specific constraints that define composition correctness predicates. *Design rule checking (DRC)* is the process by which design rules are composed and their predicates validated. A general theory for DRC is based on attribute grammars. Consider a grammar representation of model **C**:

```
C :- Base | BigI | BigD | Iadd | Idiv |
     Isub | Dadd | Ddiv | Dsub ;
```

A sentence of this grammar defines a syntactically correct composition of terms (units of **C**). Like any grammar, some sentences are semantically valid while others are not. To weed out incorrect sentences, a grammar is augmented with variables called *attributes*. Conditions for correct sentences (or correct compositions) are predicates defined over these

4. So why not just define one layer to represent both? This could be done with our current tools, as they are preprocessors. In future tools, these files will be different, because the types of variables for **e1**—**e3** will need to be explicitly declared. When this occurs, the corresponding files will indeed be different.

attributes. Two different sets of attributes are used: those whose values propagate from right-to-left (called *inherited* or *flowleft* attributes), and those whose values propagate from left-to-right (called *synthesized* or *flowright* attributes).

AHEAD provides a tool for design rule checking called **drc**. A **drc** specification lists all attributes that are externally or locally defined to that layer, followed by a set of preconditions (**requires**) and postconditions (**provides**) for inherited (**flowleft**) attributes and synthesized (**flowright**) attributes. **drc** attributes can be of two types: integer-ranges (**Int**) or a null-valued boolean (**Bool**).

The legal set of sentences/compositions of **C** is depicted using choice notations in Figure 3. Either **BigInteger** or **BigDecimal** is chosen and at least one operation is selected followed by **BigI•Base** or **BigD•Base**. Question: how can these constraints be expressed as design rules?

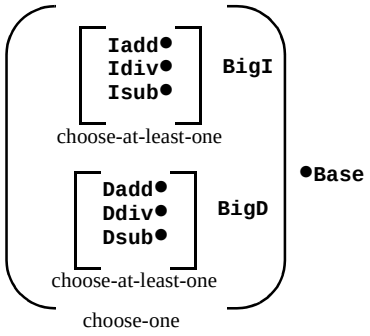



**Figure 3. Legal sentences/compositions of C**

There are many ways: here is one that uses only **Bool** attributes. The initial value of **drc** attributes is 'unknown', and can be subsequently set to 'true' or 'false'.

We use three attributes: **start**, **type**, and **atLeastOneOp**. **start** and **type** are **flowright** attributes — their values are propagated right-to-left. **start** is true when the **Base** layer is present in an equation. **type** is true when either the **BigI** or **BigD** layer is present in an equation.

**atLeastOneOp** is an **flowleft** attribute — its value is propagated from left-to-right. **atLeastOneOp** is true when at least one operation layer (e.g., **Iadd**, **Idiv**, **Dadd**, or **Ddiv**) is present in an equation. All three attributes are defined in the **Base** layer's design rule file (Figure 4a).

Preconditions on the usage (or position) of a layer in an equation are defined by **requires** predicates. The **Base** layer has a **requires** predicate which states that at least one operation unit appear in an equation (Figure 4a). Postconditions of a unit are defined by **provides** predicates. **Base** has a **provides** predicate that initializes the **start** and **type** attributes to 'true' and 'false' respectively.

The preconditions for using **BigI** are listed in Figure 4b. **BigI** is correctly used if **start** is true and **type** is false. (This constraint forces **Base** to be "right" of **BigI**). A postcondition for **BigI** is that **type** is true. The design rule file for **BigD** is the same as that for **BigI**.

The design rule files for operation units are essentially the same (Figure 4c). The precondition for their use is that both **start** and **type** be asserted (meaning that **Base** and either **BigI** or **BigD** are to the "right" of the operation in an equation). The postcondition is that **atLeastOneOp** is true, meaning that there is at least one operation in the equation.

A feature of **drc** specifications is the "**single**" modifier, which all design rule files in **C** have. **single** means that a layer can appear at most once — a single time — in an equation. So compositions like **BigI•BigI** are flagged as errors. The "**constant**" modifier tells **drc** that the layer is a constant, not a refinement. The **Base** file is a constant (Figure 4a); the other units of **C** are refinements.

## Model Exercises

[5] Our model is not exactly complete. It admits incorrect compositions like: **Ddiv•BigI•Base**. How would you extend the design rule files to preclude this possibility?

[6] It is often the case that refinements appear in pairs. Suppose if the addition operation appears in an equation, so too must the subtraction operation. The operations don't have to be adjacent — if addition is added to an equation, eventually subtraction will be too, and vice versa. How would you extend the design rule files to express this possibility?

```
single constant layer;

flowleft Bool start;
flowleft Bool type;
flowright Bool atLeastOneOp;

provides flowleft start and !type;
requires flowright atLeastOneOp;
```

(a) base/rules.drc

```
single layer;

extern flowleft Bool start;
extern flowleft Bool type;

requires flowleft start and !type;
provides flowleft type;
```

(b) BigI/rules.drc

```
single layer;

extern flowright Bool atLeastOneOp;
extern flowleft Bool type;
extern flowleft Bool start;

requires flowleft type and start;
provides flowright atLeastOneOp;
```

(c) Iadd/rules.drc

**Figure 4. Design Rule Files**

```
single layer i3;

flowright Bool atLeastOneOp;
flowleft Bool start;
flowleft Bool type;

// externally defined attributes

provides right atLeastOneOp;
provides left start and type;
```

Figure 5. i3/rules.drc

[7] Extend the DRC specifications so that calculators have a minimal number of operations (not just, say, division).

## Tool Exercises

To each subdirectory of **C**, add the appropriate design rule file. Here is a part of the **C** directory structure:

```
C/Base/calc.jak     // see Figure 1a
C/Base/rules.drc    // see Figure 4a
C/BigI/calc.jak     // see Figure 1b
C/BigI/rules.drc    // see Figure 4c
C/Iadd/calc.jak     // see Figure 1d
C/Iadd/rules.drc    // see Figure 4c
...
```

When **composer** evaluates an equation, the corresponding Jak and drc files in each layer are composed. Thus, using the same call to **composer** as before:

```
> cd C
> composer --target=i3 Base BigI Idiv Iadd
```

produces a directory **i3** with two files: **calc.jak** (see Figure 2) and **rules.drc** (see Figure 5). Note that the generated DRC file has no **requires** statements. Generally if there are requirements, this means that the synthesized program is incomplete — other layers (units) must be composed to form a complete program. For example, the composition **BigI•Base** is incomplete, because it lacks an operation layer. When **composer** is run on this equation, a warning is generated by **drc**:

```
> composer --target=i3prime Base BigI

drc Warning: rules.drc: equation i3prime is
incomplete: non-null requirements present

requires left atLeastOneOp
composer: 1 warning occurred
```

The corollary is also true: a lack of **requires** statements usually means that the resulting Jak files can be translated to Java using the **jak2java** tool.

As an exercise, try building other equations and examine/understand its results.

# 4 Other ATS Tools

There are quite a few ATS tools that you can use. You have seen **composer**, **jak2java**, **jampack**, and **drc**. Now let's look at **mixin**, **unmixin**, and **reform**.

## 4.1 mixin

**composer** invokes a set of tools when features are composed. The **drc** tool composes drc files, and the **jampack** tool composes Jak files.

**mixin** is another tool that can compose Jak files. Edit the **unit.file.jak** line in the **composer.properties** file to be:

```
unit.file.jak : MixinFileUnit
```

This is the default setting for **unit.file.jak**. If **composer** doesn't see a **composer.properties** file, it uses **mixin** to compose Jak files.

Let's re-evaluate the **i3** equation to see how **mixin** works:

```
> cd C
> composer --target=i3 Base BigI Idiv Iadd
```

This is the same command as before. However, the **calc.jak** file that is produced is quite different and is shown in Figure 6.

The idea behind **mixin** is simple: each refinement is mapped to class in an inheritance/refinement chain. Each class (or interface) is prefaced by a **SoUrCe** statement which indicates the name of the feature and the actual file from which the class was derived. Thus, in Figure 6 four Jak files were composed to yield the **calc** class; this class is the terminal class of the refinement chain. All other classes are abstract — meaning that they can't be instantiated and whose purpose is only to contribute members to the final class in the chain. Note that class names are *mangled* (i.e., by appending **$$<layerName>**) to make them unique.

The intent of **mixin** and **jampack** is that you can use either tool to compose Jak files. As you'd expect, the programs of Figure 2 and Figure 6 are functionally equivalent.

Both **mixin** and **jampack** can compose files that they themselves have produced. That is, a **jampack**-produced Jak file can be composed with another **jampack**-produced Jak file. The same holds for **mixin**. Because **jampack**-produced Jak files have the same format as uncomposed Jak files, **mixin** can compose files produced by **jampack**. However, the reverse is not true: **jampack** cannot compose **mixin**-produced files.

```
layer i3;

import java.math.BigInteger;

SoUrCe RooT base "../base/calc.jak";
abstract class calc$$base {}

SoUrCe  BigI "../BigI/calc.jak";
abstract class calc$$BigI extends  calc$$base {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;

    void add() {
        e0 = e0.add( e1 );
        e1 = e2;
    }

    void clear() {
        e0 = e1 = e2 = zero;
    }
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }

    void enter( String val ) {
        e2 = e1;
        e1 = e0;
        e0 = new BigInteger( val );
    }

    String top() {
        return e0.toString();
    }
}

SoUrCe  Iadd "../Iadd/calc.jak";
abstract class calc$$Iadd extends  calc$$BigI {
    // adds BigIntegers
    void add() {
        // adds BigIntegers
        e0 = e0.add( e1 );
        e1 = e2;
    }
}

SoUrCe  Idiv "../Idiv/calc.jak";
class calc extends  calc$$Iadd {
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }
}
```

**Figure 6. mixin-produced .jak file**

## 4.2 unmixin

So why use **mixin**? Why not always use **jampack**? Consider a typical debugging cycle: you compose files, use **jak2java** to translate Jak files to Java files, compile and run the Java files to discover bugs. The original Jak files are patched and the cycle continues. You will see that knowing what feature files to update won't always be easy — and the problem becomes worse as the number and size of the Jak files increases.

Because **mixin** preserves refinement boundaries, it is easy to know what feature to update. In fact, with **SoUrCe** statements, the propagation of changes can be done automatically. That's the purpose of **unmixin**. The idea is that you compose a bunch of Jak files, edit the *composed* files, and run **unmixin** on the edited files to back-propagate the changes to the original feature files. For example, suppose we add a comment to the bottom-most class in the refinement chain of Figure 6:

```
SoUrCe  Idiv "../Idiv/calc.jak";
class calc extends  calc$$Iadd {
   void divide() {
      // divide and pop stack
      e0 = e0.divide( e1 );
      e1 = e2;
   }
}
```

By running **unmixin**, this change is propagated back to the **Idiv/calc.jak** file:

```
> cd C
> unmixin calc.jak
```

See for yourself that the change was indeed made. Here are things to remember about **unmixin**:

- it can take any number of Jak files on its command line,
- the contents of the class or interface in the command-line file will replace the contents of the class or interface in the original file,
- **implements** declarations are also propagated, and
- *don't change the contents of the* **SoUrCe** *statements!*

**unmixin** updates the original uncomposed files only if changes to its composed counterpart has been updated.

## 4.3 reform

**reform** is a pretty-printing tool that will format unruly Jak files (and Java files!) and make them unbelievably beautiful. Consider the 1-line **calc.jak** file:

```
refines class calc { void divide() { e0 =
e0.divide( e1 ); e1 = e2; } }
```

By running:

```
> reform calc.jak
```

**reform** copies the original file into **calc.jak~** and updates **calc.jak** to be:

```
refines class calc {
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }
}
```

# 5 More Features of Jak Files

There are three additional features of the Jak language that you should be aware: Super references, refinement of constructors, and local ids.

## 5.1 The Super Construct

To reference a method **`m(int x, float y)`** of a superclass in Java, you write:

```
super.m(x,y);
```

In Jak files, use the **`Super`** construct instead:

```
Super(int,float).m(x,y);
```

**`Super(<argument types>)`** prefaces a **`Super`** call and lists the argument types of the method to be called. Consider the class **`foo`** and a refinement:

```
class foo {
   void dosomething() { /*code*/ }
}

refines class foo {
   void dosomething() {
      /* more before */
      Super().dosomething();
      /* more later */
}
```

In this example, the **`Super`** references the **`dosomething()`** method prior to its refinement. A **`jampack`** composition of these definitions yields:

```
class foo {
   final void dosomething$$one() { /*code*/ }

   void dosomething() {
      /* more before */
      dosomething$$one();
      /* more later */
   }
}
```

You see that the original **`dosomething()`** method is present in **`foo`**, except that it has been renamed, along with its references. The corresponding **`mixin`** composition is:

```
SoUrCe ...;
abstract class foo$$one {
   void dosomething() { /*code*/ }
}

SoUrCe  ...;
class foo extends  foo$$one  {
   void dosomething() {
      /* more before */
      Super().dosomething();
      /* more later */
   }
}
```

When **`jak2java`** translates the above, **`Super(...)`** references are replaced by "**`super`**". In general, always use the **`Super(...)`** construct to reference superclass members; ATS tools do not recognize "**`super`**".

## 5.2 Refining Constructors

A constructor is a special method and to refine a constructure requires a special declaration in Jak files. Consider the following file that declares a constructor:

```
class test {
   int y;
   test() { y = -1; }
}
```

A refinement of **`test`** and its constructor is:

```
refines class test {
   int x;
   refines test() { x = 2; }
}
```

where "**`refines <constructor>`**" is the Jak statement that declares a refinement of a particular constructor. The **`jampack`** composition of these files is:

```
class test {
   int y;
   int x;
   test() { { y = -1; } { x = 2; } }
}
```

That is, the actions of the original constructor are grouped into a block and are performed first, then the actions of the constructor refinement are grouped into a block and performed next. The semantically equivalent **`mixin`** composition is:

```
SoUrCe ...;
abstract class test$$t1 {
   int y;
   test$$t1() { y = -1; }
}

SoUrCe ...;
class test extends  test$$t1  {
   int x;
   refines test() { x = 2; }
}
```

## 5.3 Local Ids

Variables that are local to a layer/feature are common. Such variables are used only by the layer itself, and are not to be exported or referenced by other layers.

Suppose a class **bar** declares a local variable **x**, and a refinement of **bar** declares a local variable, also named **x**:

```
class bar {
   int x;
}
```

```
refines class bar {
   float x;
}
```

**jampack** is smart enough to alert you that multiple definitions of **x** are present; **mixin** isn't that smart — and you will discover the error when you compile the translated Java files and see there are multiple definitions of **x**.

The way to fix this is by using a **Local_Id** declaration. This declaration lists the set of identifiers (i.e., variable names and method names) that are *local* to a feature; ATS tools will mangle their names so that they will always be unique. So a better way to define the above is:

```
Local_Id x;

class bar {
   int x;
}
```

```
Local_Id x;

refines class bar {
   float x;
}
```

The **jampack** and **mixin** compositions are shown below:

```
class bar {
   int x$$one;
   float x$$two;
}
```

```
SOURCE ...;
abstract class bar$$one {
   int x$$one;
}

SOURCE  ...;
class bar extends  bar$$one  {
   float x$$two;
}
```

where local names are replaced with their mangled counterparts so that their names no longer conflict.

## 6 A More Complex Example

Consider model **L**, which defines a set of programs that implement linked lists:

```
L = { sgl, dbl, sgldel, dbldel }
```

The lone constant is **sgl** which contains a pair of classes, **list** and **node**, that implement a bare-bones singly-linked list (Figure 7a-b).

A refinement of **sgl** is **dbl**, which converts the program of **sgl** into a doubly-linked list. **dbl** is a "cross-cut" that augments the **node** class with a **prior** pointer, adds a **last** pointer to the **list** class, and refines the **list insert** method so that the constants of the **last** and **prior** pointers are consistent (Figure 7c-d).

The composition **both = dbl•sgl** yields the doubly-linked list program of Figure 8. The code indicated in red originates from the **dbl** refinement.

```
class list {
   node first = null;
   node last = null;

   final void insert$$sgl( node n ) {
      n.next = first;
      first = n;
   }

   void insert( node n ) {
      if (last == null)
        last = n;
      if (first != null)
        first.prior = n;
      insert$$sgl.insert(n);
     n.prior = null;
   }
}
```

(a) L/both/list.jak

```
class node {
   String constant;
   node   next = null;
   node prior = null;
}
```

(b) L/both/node.jak

**Figure 8. Composition dbl•sgl**

```
class list {
   node first = null;

   void insert( node n ) {
      n.next = first;
      first = n;
   }
}
```

(a) L/sgl/list.jak

```
class node {
   String constant;
   node   next = null;
}
```

(b) L/sgl/node.jak

```
refines class list {
   node last = null;

   void insert( node n ) {
      if (last == null)
        last = n;
      if (first != null)
        first.prior = n;
      Super(node).insert(n);
     n.prior = null;
   }
}
```

(c) L/dbl/list.jak

```
refines class node {
   node prior = null;
}
```

(d) L/dbl/node.jak

**Figure 7. The sgl and dbl Layers**

Now suppose we want to enhance the design of our list programs by adding a **delete** method. **sgldel** does exactly this for singly-linked lists: it adds a **delete** method to the list class (Figure 9a).

We can use sgldel in a composition **slist** that defines a singly-linked list with both insert and delete methods:

```
slist = sgldel • sgl
```

To create a doubly-linked list that has both insert and delete methods requires a new refinement **dbldel** (see Figure 9b). **dbldel** converts the singly-linked list deletion algorithm of **sgldel** to a doubly-linked list deletion algorithm by replacing (or overriding) the **findAndDelete** method.

The following equations yield identical programs for inserting and deleting elements from a doubly-linked list. The reason why they are equivalent is that the refinements **dbl** and **sgldel** are independent of each other, and thus can be composed in any order.

```
dlist = dbldel•dbl•sgldel•sgl          (9)

      = dbldel•sgldel•dbl•sgl          (10)
```

## Model Exercises

[8] Suppose other operations for traversing the list were added. How would this impact model **L**? What about the operation reverse, which reverses the order in which nodes are listed?

```
refines class list {

   void delete( node n ) {
      if (n == first) {
        first = first.next;
      }
      else
        findAndDelete(n);
   }

   void findAndDelete(node n) {
      node prev = first;
      while (prev != n)
         prev = prev.next;
      prev.next = n.next;
   }
}
```

(a) L/sgldel/list.jak

```
refines class list {

   void findAndDelete(node n) {
      if (n.prior != null)
         n.prior.next = n.next;
      if (n.next != null)
         n.next.prior = n.prior;
   }
}
```

(b) L/dbldel/list.jak

**Figure 9. The sgldel and dbldel Layers**

[9] Suppose the "ordering" feature is added to a list, meaning that nodes have keys and nodes are maintained in ascending key order. How would this feature impact **L**?

[10] Consider a "monitor" feature, which precludes more than one thread to access a list at a time. How would this feature impact **L**? How would it be defined?

## Tool Exercises

The directory structure for **L** is:

```
L/sgl/list.jak      // see Figure 7a
L/sgl/node.jak      // see Figure 7b
L/dbl/list.jak      // see Figure 7c
L/dbl/node.jak      // see Figure 7d
L/sgldel/list.jak   // see Figure 9a
L/dbldel/list.jak   // see Figure 9b
```

The files of Figure 8 are the result of evaluating the equation **both = dlb•sgl** using the **composer** tool:

```
> cd L
> composer --target=both sgl dbl
```

The generated directory structure is:

```
L/both/list.jak     // see Figure 8a
L/both/node.jak     // see Figure 8b
```

## 6.1 Adding Design Rules

Once again, not all compositions of units of **L** are semantically correct. The legal compositions are:

```
sgl
dbl•sgl
sgldel•sgl
dbldel•sgldel•dbl•sgl
dbldel•dbl•sgldel•sgl
```

What are the design rules for **L** that would admit only these compositions? The rules are not difficult to express if disjunctions are allowed.[5] But unfortunately, the **drc** tool currently only admits conjunctive predicates, thus making it impossible to express such rules. At least at first glance.

There is a deeper relationship among the refinements of **L** that we have not yet captured. By making this relationship explicit, we can express the design rules of **L** using elementary conjunctive predicates. The idea that is missing is called *origami*, the topic of the next section.

5. An equation is legal if **sgl** is the left-most unit, *or* if **dbl** is the left-most unit and is followed by **sgl**, *or* if **sgldel** is the left-most unit and is followed by **sgl**, *or* if **dbldel** is the left most unit and is followed by either **dbl** or **sgldel** (in either order) followed by **sgl**.

## 6.2 Origami

Consider the following incorrect compositions:

```
error1 = dbl•sgldel•sgl
error2 = dbldel•sgldel•sgl
```

Both define programs that are partially and thus incorrectly implemented. **error1** is a program whose **insert** method works on a doubly-linked list, but whose **delete** method works only on a singly-linked list. **error2** is a program whose **insert** method works on a singly-linked list, but whose **delete** method works for a doubly-linked list.

The problem here is that if a data structure is extended (i.e., a singly-linked list becomes doubly-linked), then *all* of its operations should be updated to maintain the consistency of this extension, and not just some. That is, if a singly-linked list has both insert and delete operations, when the structure becomes doubly-linked, both operations must be updated to work on doubly-linked lists. Equivalently, if a refinement adds a new method to a data structure, then that method must work for that data structure and not some other structure.

Although this is an elementary example, it is representative of a large class of problems in FOP, namely that a model defines a set of refinements that are not truly independent and must be applied in groups in a lock-step — or all or nothing — manner. Whenever you notice this phenomena, realize that these groups represent "higher-level" features.

Here's a useful technique for understanding this problem. Create a matrix, called an *origami matrix*, where rows represent operations (**insert**, **delete**), and columns represent structure variants (**singleLink**, **doubleLink**). Entries of this matrix are the refinements of **L** (see Table 1). This matrix can be extended to handle other operations (sort, find) and other structure variants (ordered-lists, monitors, etc.).

> Note: what we have done is to identify the orthogonal "higher-level" features as 'data structure operations' and 'data structure variants'.

| | doubleLink | singleLink |
|---|---|---|
| insert | dbl | sgl |
| delete | dbldel | sgldel |

**Table 1 Origami Matrix for L**

Suppose the rows of this matrix are composed (or *folded* — hence the name "origami"), where the corresponding entries in each column are composed:

| | doubleLink | singleLink |
|---|---|---|
| delete•insert | dbldel•dbl | sgldel•sgl |

**Table 2 Row-Composed Origami Matrix**

Study the entries of Table 2. Consider the entry in the **singleLink** column: **sgldel•sgl** defines a singly-linked program **S** that has both an **insert** and **delete** method. The entry in the **doubleLink** column, **dbldel•dbl**, defines a refinement of **S** that converts its insert and delete methods to work on a doubly-linked list. Thus by composing the **delete** row with the **insert** row of Table 1, we synthesize a data structure that has multiple methods, and a refinement of that data structure that consistently updates these methods. This interpretation holds if more rows (operations) or more list features (columns) are added.

The columns of Table 2 can be composed to yield a 1×1 matrix whose entry is an expression that defines a doubly-linked list with insert and delete methods (Table 3). This expression is identical to equation **(9)**.

| | doubleLink•singleLink |
|---|---|
| delete•insert | dbldel•dbl•sgldel•sgl |

**Table 3 A Completely Folded Matrix**

Now instead of composing rows of Table 1, let's compose the columns, where corresponding entries in each row are composed:

| | doubleLink•singleLink |
|---|---|
| insert | dbl•sgl |
| delete | dbldel•sgldel |

**Table 4 Column Composed Origami Matrix**

The entry in the **insert** row, **dbl•sgl**, defines program **D** that implements a doubly-linked list with an **insert** method. The entry in the **delete** row, **dbldel•sgldel**, defines a refinement of **D** that adds a **delete** method. By composing the columns of Table 1, we have synthesized a data structure with a single (**insert**) method, and a refinement that adds a **delete** method to this structure. Again, this interpretation holds if we add more rows (methods) or more columns (features) to Table 1. By folding the rows of Table 4 yields a 1×1 matrix whose entry is equation **(10)**. As a general rule, as long as the order in which rows and columns (that is, 'data structure operation' features or 'data structure variant' features) are composed are legal (where legality is a topic of an upcoming section), the resulting equations in a fully-folded matrix are equivalent.

Origami matrices capture fundamental relationships among groups of refinements: to build consistent and correct programs, it is often necessary to apply an entire set of refinements at once. A matrix representation of these relationships works because the set of features along one dimension are *orthogonal* to those of another. In our example, the set of methods that can be used with a data structure is orthogonal to the set of data structure variants.

Although this is a simple example, origami applies at much greater levels of granularity. For example, ATS has five tools — including **jampack**, **mixin**, and **unmixin** — each having over 30K LOC, and totalling over 150K LOC. These tools are synthesized by folding a 3-dimensional (8×6×8) Origami matrix.

## 6.3 Equation Files

How are Origami matrices represented in AHEAD? Before we can answer this question, we need to introduce some concepts.

Typing in equations on the command line to **composer** can be tedious, particularly if the equations are more than a few terms. **composer** takes an alternative specification, called an *equation file*, which is a list of units to compose. The order in which the units are listed is from inside-out, and the name of the equation is the name of the equation file.

For example, the equation **A = B•C** would be represented by the equation file **A.equation** whose contents is:

```
# inside-to-outside order!
C
B
```

Where any line beginning with # is a comment. Like any other artifact, equation files can be composed. File **A.equation** above is a "constant". An equation file that is a refinement has the special term **super** as one of its units. A refinement of **A.equation** that puts **E** before **C** and **F** after **B** is **R.equation**:

```
E
super
F
```

A composition of the above files is expressed by:

```
> composer --target=c.equation \
           A.equation R.equation
```

which yields file **c.equation** with contents:

```
E
C
B
F
```

Now onto metamodels.

## 6.4 Metamodels and Origami Matrices

A *metamodel* is a model whose instances are themselves models. Consider model **M**, which has units **a**, **b**, and **c**:

```
M = { a, b, c }
```

Now consider metamodel **MM**, which has also has three units, each being a collective with a single unit:

```
MM = { AAA, BBB, CCC }
   = { {a}, {b}, {c} }
```

A model can be synthesized by composing metamodel units. The **MM** equation for model **M** is:

```
M = AAA•BBB•CCC
```

In this particular case, because there are no units in common with **AAA**—**CCC**, composition reduces to set-union. The interesting thing about metamodels is that they are identical to models. That is, a model or metamodel is a set of units, where each unit may be a collective. Further, the composition operator for units of metamodels is the same operator for units of models.

An origami matrix is a 2-dimensional (and in general, an k-dimensional) array of units. A collective is a hierarchy. So to represent matrices in AHEAD, we need to encode a matrix as a tree. For a 2-dimensional matrix we can decompose it first into rows, and then each row into columns. Another way is to organize by columns first, and then rows. Figure 10 shows these embeddings for an $n \times m$ matrix **O** where $\mathbf{O}_{ij}$ denotes the row $i$ column $j$ element of **O**.

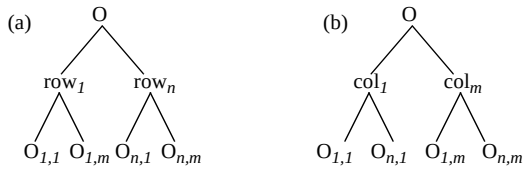



**Figure 10. Matrix Embeddings in Trees**

## 6.5 Representing Origami Matrices

Given the ideas of equation files and metamodels, we're ready to see how origami matrices are represented.

First, consider a row-dominant representation of a matrix. Figure 11a shows our example origami matrix, where the matrix entries are equation files that have the same name (**eqn.equation**). Entry subscripts denote (to us) their true identity. Figure 11b is the corresponding row-oriented metamodel; Figure 11c is its AHEAD directory structure. Figure 11d-g are the contents of the equation files.

(a)

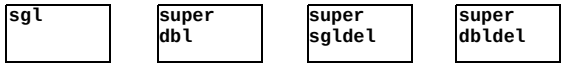
| | single | double |
|---|---|---|
| insert | eqn.equation$_{sgl}$ | eqn.equation$_{dbl}$ |
| delete | eqn.equation$_{sgldel}$ | eqn.equation$_{dbldel}$ |

(b)
```
Origami = { insert, delete }

insert = { single_i, double_i }
double = { single_d, double_d }

single_i = { eqn_sgl }      single_d = { eqn_sgldel }
double_i = { eqn_dbl }      double_d = { eqn_dbldel }
```

(c)
```
Origami/insert/single/eqn.equation // Figure 11d
Origami/insert/double/eqn.equation // Figure 11e
Origami/delete/single/eqn.equation // Figure 11f
Origami/delete/double/eqn.equation // Figure 11g
```




(d) (e) (f) (g)

**Figure 11. Row-Dominant Embedding of a Matrix**

Why do we use this particular representation of a matrix? Why use equation files, rather than embedding the actual feature directories themselves? The answer: convenience. Try to create such a hierarchical directory yourself, where instead of equation files, you have feature directories. It's hard to navigate such a directory structure, let alone maintain it. The simpler the representation the better. So it is common that we have a flat model directory (where features are immediate subdirectories), and a separate origami directory which defines the 2D relationship among features using equation files.

Doing this improves matters slightly. Fortunately, there is a special AHEAD tool, called **obe** (short for Origami BrowsEr), that allows you to create, populate, document, and compose origami matrices in short order. I'll leave it as an exercise for you to read about **obe** and how to use it.

## Model Exercises

[11] Expand the origami matrix to handle more data structure operations and variants. Consider one variant to be element compression — elements are stored in a compressed manner, but are inserted and retrieved in an uncompressed manner.

[12] Recall model **C** from Section 3. Units of **C** contained both Jak files and DRC files. Create a metamodel **MC** that has two units, **Code** and **Rules**. The **Code** unit is a collective that contains only Jak files, the **Rules** unit is also a collective that contains the corresponding DRC files. Show that model **C** can be synthesized by either **Code•Rules** or **Rules•Code**.

## Tool Exercises

Once the origami matrix is set up, you can use **obe** to compose rows and columns, or you can do it manually, as shown below.

To fold a 2-dimensional matrix, you need to invoke **composer** twice: once to compose rows and a second time for columns. (For a k-dimensional matrix, we would invoke **composer** k times). So to produce the AHEAD equivalent of Table 2, we compose the rows of the **origami** model to produce model **Table2** = **delete•insert**:

```
> cd origami
> composer --target=Table2 insert delete
```

The resulting model **Table2** is depicted in Figure 12a, and its synthesized AHEAD directory structure in Figure 12b, and the contents of the equation files in Figure 12c-d.

(a)
```
Table2 = { single, double }

single = { eqn_sgldel•eqn_sgl }
double = { eqn_dbldel•eqn_dbl }
```

(b)
```
Origami/Table2/single/eqn.equation // Figure 12c
Origami/Table2/double/eqn.equation // Figure 12d
```

(c)
```
sgl
sgldel
```

(d)
```
super
dbl
dbldel
```

**Figure 12. AHEAD Origami Metamodel**

To produce the 1×1 matrix of Table 3 or equation **(9)**, we compose the columns (named **single** and **double**) using the following command:

```
> cd Table2
> composer --target=both single double
> cd both
> jak2java *.jak
```

Another way to represent the origami matrix of Figure 11a is by columns. We leave this as an exercise.

As another exercise, implement your solution to problem [12].

## 6.6 Design Rules

Now we can address the problem of defining design rules for our origami representation of model **L**. Recall that the design rules for **L** without origami were too complicated to express using the **drc** tool. With origami, design rules become much simpler. All we need to do is to create design rules for composing "higher-level" features. That is, we need rules to define the legal orders for composing rows (i.e., data structure operations) and columns (data structure variants).

The rules for ordering rows is simple: the **insert** row is a 'constant' as it represents a base data structure (Figure 13a). The **delete** row is a 'function' that refines the **insert** row. Rule: **delete** must follow (i.e., be to the right of) **insert** (Figure 13b). The same rule can be used for the columns in our example.

```
single constant layer;

flowleft Bool start;

provides flowleft start;
```

(a) insert/rules.drc

```
single layer;

extern flowleft Bool start;

requires flowleft start;
```

(b) delete/rules.drc

**Figure 13. Dimension Design Rule Files**

Where are these files stored? The **obe** tool stores them in separate directories, one per dimension. I leave it to you as an exercise to determine where these files might be stored if you don't use **obe**.

### Model Exercises

[13] Create two different GUIs for a calculator: one uses the standard 2D keypad, a second uses text fields to enter constants and operations. A calculator will use one (but not both) of these GUIs. Operations on both GUIs are buttons. So when a calculator is extended by a new operation, its GUI will be extended also. Express this relationship between operation and GUIs as an origami matrix.

[14] As additional exercises: (1) generalize the above model that permits multiple GUIs per calculator. One idea would use tabs, one tab per different GUI. (2) implement your model.

## 7 What's Ahead0

There are lots of interesting topics and capabilities that we have yet to explore (or develop) for AHEAD. Here are just a few.

### 7.1 Extensible Languages

There are all sorts of non-Java extensions to the Jak language that we haven't talked about, including:

- *metaprogramming* — the ability to assign code fragments to variables, the ability to compose code fragments via escape substitutions, hygienic macros.
- *state machines* — an embedded DSL for supporting the definition and refinement of state machines.

More on these topics in a future paper.

### 7.2 Compiler-Compiler Tools

ATS has a sophisticated set of compiler-compiler tools that are used to (a) define base grammars, (b) define grammar refinements, and (c) to synthesize grammars by composing base grammars with refinements. Grammars are just another representation of a program, in this case, a compiler, and ATS has tools for defining and composing grammars and generating Java files from them. More on these topics in a future paper.

### 7.3 Generating and Optimizing MakeFiles

The idea of modules as hierarchical collections of related artifacts is very powerful. A paradigm of AHEAD that we have explored so far is that of *composition*: that artifacts of a system can be composed from previously defined artifacts. But there is another way in which system artifacts can be produced: by *derivation*. For example, Java files can be produced from Jak files by the **jak2java** tool; class files can be produced from Java files by the **javac** compiler, and so on. A general paradigm is depicted in Figure 14: an artifact can be produced by first composing it from more elementary artifacts, followed by a derivation. Or equivalently, it can be produced by deriving a set of artifacts from more elementary artifacts first, and then composing the derived representations[6]. This leads to the following fundamental distributive algebraic relationship **(11)**.

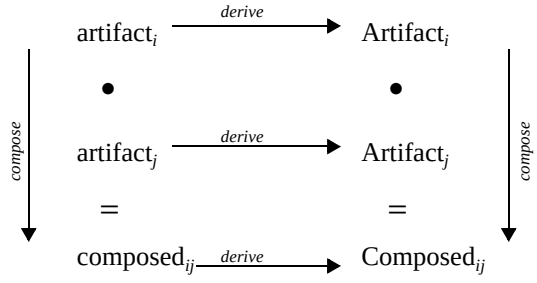



**Figure 14. Compose vs. Derive**

$$\mathbf{derive}(\ \mathbf{artifact}_i \bullet \mathbf{artifact}_j\ ) = \mathbf{derive}(\ \mathbf{artifact}_i\ ) \bullet \mathbf{derive}(\ \mathbf{artifact}_j\ ) \quad \mathbf{(11)}$$

Ultimately, what we want is to specify an entire system — all of its composed and derived representations — as a set of equations. Although ATS does not yet have such a tool, one can imagine a specification like:

```
Using L;

i3 = javadoc( javac( jak2java(
           sgldel( sgl ) ) ) );
```

6. Figure 14 can be generalized further, so that multiple output artifacts can be derived from a single input artifact, and vice versa.

Where the **`Using`** clause will tell this tool that **`sgldel`** and **`sgl`** are units in model **`L`**, and by inference, **`composer`** should be used to compose them. The resulting module will have Drc and Jak files. The **`jak2java`** tool, when applied to a collective/directory, will translate all Jak files to Java files and add them to that directory. Similarly, the **`javac`** operator will compile all Java files in a collective to their class file counterparts. The **`javadoc`** operator will generate JavaDoc HTML pages from the generated Java files, and so on.

The lesson that we learned from relational query optimizers is that given an equational form of a program specification, equations can be optimized. In this particular example, there really isn't anything to be optimized. There is, though, a particular sequencing of the application of the **`javadoc`**, **`javac`**, and **`jak2java`** operators that must be imposed. (In fact, this really is the only legal ordering of these operations for this equation). So notions of design rules apply to tool operators. But as equations become more complicated, there is the possibility of optimization. In some of our larger examples using origami, generating common subexpressions among different sets of tools arises. Evaluating common subexpressions once, and not many times, is an important optimization that a tool should be able achieve automatically.

The big picture is depicted below. Given an equational representation of a system that specifies both the artifacts that are to be composed and those that are to be derived, a tool will expand the equations and perform optimizations to synthesize the resulting system in the most efficient manner. The tool will then produce an optimized set of equations, and a generator will translate these equations into a *makefile* — a low-level implementation of a functional-like language that will efficiently execute equational specifications.

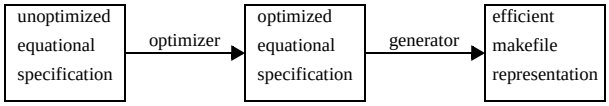



**Figure 15. Generation and Optimization of MakeFiles**

## 7.4 Typing Programs

As mentioned earlier, refinements are functions that appear untyped. In fact, function inputs and outputs have definite constraints. Our tools simply have assumed that the correct types are both being input and output. In general, this is bad assumption.

Question: how does one type a program? Should Java interfaces be used? How does typing generalize to, say, drc files? How are drc files typed? What is a general mechanism for typing arbitrary artifacts and their refinements? We have the outlines of a solution for AHEAD, but at this time, there is no tool support.

# 8 Conclusions

Expressing programs as compositions of higher-level abstractions is very appealing. Abstractions, which correspond to GenVoca/AHEAD constants or functions, have a straightforward representation in programming languages and DSLs that have been extended to support step-wise refinement. On a broader scale, it is intuitive to envision that Software Engineering is about the tools and techniques for program design and manipulation. It is appealing to see that there is a simple algebraic foundation for supporting this viewpoint when SE is viewed from an FOP perspective: namely, programs are values and functions (or operators) transform programs in standardized, domain-prototypical ways.

By elevating the level of abstraction, we can see and manipulate "large-scale" or "architectural" relationships among modules called "refinements" that comprise a program — relationships that previously were implicit and hard-wired. The modularity that we expose through FOP and ATS is different than conventional notions, and it is through this novelty that we gain considerable power. By expressing designs as models (algebras), we create designs for families of programs called product-lines, and create designs for programs that are architecturally extensible (i.e., that can be redefined by new compositions of refinements).

This paper has explored basic concepts of FOP and a (small) subset of the tools of the AHEAD tool suite. In future papers, we will see further evidence of the power and generality of AHEAD and its tools.

**Acknowledgements**. I thank Stijn Coene for identifying errors in this text.

# 9 Suggested Reading

[1] Don Batory, Jacob Neal Sarvela, Axel Rauschmayer, "**Scaling Step-Wise Refinement**", 2003 International Conference on Software Engineering. First paper on the AHEAD model.

[2] Don Batory, Roberto Lopez-Herrejon, Jean-Phillipe Martin, "**Generating Product-Lines of Product-Families**", Automated Software Engineering Conference, 2002. A sophisticated example of the origami model.

[3] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder, "**Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study**", ACM Transactions on Software Engineering and Methodology, Vol 11#2, April 2002, 191-214. A sophisticated product-line that requires both refinements and embedded DSLs.

[4] Don Batory, Gang Chen, Eric Robertson, and Tao Wang, **Design Wizards and Visual Programming Environments for GenVoca Generators**, *IEEE Transactions on Software Engineering*, May 2000, 441-452. Explains relationship between automatic programming and GenVoca equation optimization.

[5] Don Batory and Bart J. Geraci. **Composition Validation and Subjectivity in GenVoca Generators**, *IEEE Transactions on Software Engineering* (special issue on Software Reuse), February 1997, 67-82. Defines design rule checking.

[6] Roberto E. Lopez-Herrejon and Don Batory, **A Standard Problem for Evaluating Product-Line Methodologies**, *Third International Conference on Generative and Component-Based Software Engineering (GCSE 2001), September 9-13, 2001 Messe Erfurt, Erfurt, Germany.* A simple product-line defined using the GenVoca model.