ACM SIGSOFT 2003

Refinements and Multi-Dimensional Separation of Concerns

Don Batory, Jia Liu, Jacob Neal Sarvela
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{batory, jliu, sarvela}@cs.utexas.edu

ABSTRACT[1]

Step-wise refinement (SWR) asserts that complex programs can be derived from simple programs by progressively adding features. The length of a program specification is the number of features that the program has. Critical to the scalability of SWR are multi-dimensional models that separate orthogonal feature sets. Let n be the dimensionality of a model and k be the number of features along a dimension. We show program specifications that could be $O(k^n)$ features long have short and easy-to-understand specifications of length $O(kn)$ when multi-dimensional models are used. We present new examples of multidimensional models: a micro example of a product-line (whose programs are 30 lines of code) and isomorphic macro examples (whose programs exceed 30K lines of code). Our work provides strong evidence that SWR scales to synthesis of large systems.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: methodologies, tools; D.2.10 [**Design**] methodologies, specification; D.2.11 [**Software Architectures**]: data abstraction; D.2.13 [**Reusable Software**] domain engineering, reusable libraries; I.2.2 [**Automatic Programming**]: program synthesis, program transformation.

General Terms

Design

Keywords

multidimensional separation of concerns, refinements, AHEAD, feature-oriented programming, Origami, program synthesis, GenVoca.

1. This work was supported in part by the U.S. *Army Simulation and Training Command (STRICOM)* contract N61339-99-D-10.

1 Introduction

Multi-Dimensional Separation of Concerns (MDSOC) is an approach to simplify the specification, development, and design of software [14][20][15]. MDSOC asserts that a program can be decomposed or modularized in many different ways: by classes, by features, by aspects, by functions, etc. Each partitions the space of primitive software artifacts, called *units*, in a different way. Thus, partitioning software by features "cross-cuts" a partitioning by classes, and vice versa. The same applies to other modularizations. MDSOC advocates that modularity can be understood by multi-dimensional spaces of units, where dimensions represent different modularizations (e.g., classes), and units along a dimension are particular instances of that dimension's modularity (e.g., specific classes). Further, no preference is given to a particular dimension: all are treated equally.

Our research is in software product-lines and the synthesis of programs that are members of a product-line [2]. The hallmark of product-lines is its use of features to describe and distinguish product-line members [9][11][22]. A *feature* is a characteristic that programs of a product-line can share; distinct programs in a product-line are described by distinct combinations of features.

Our twist on product-lines explores feature modularity. When a program is described by features, we synthesize that program by composing modules that implement those features. The basis of our approach is step-wise refinement, which asserts that a complex program can be synthesized from a simple program by progressively adding features [8][6].

It would seem that our view of modularity is the antithesis of MDSOC, which explicitly advocates multiple modularizations and criticizes the use of only one. In reality, these approaches are actually much closer than they appear. An interesting overlap occurs when software can be modularized by orthogonal sets of features. In a 2-dimensional case, both dimensions represent feature sets, where a feature of a column dimension "cross-cuts" the units that comprise features of the row dimension, and vice versa.

Our preliminary recognition of this connection with MDSOC was reported in [3]. Our understanding of this relationship has evolved substantially since, and in this paper we present new examples and a more general interpretation of MDSOC in the context of step-wise refinement. We demonstrate the generality of the marriage of these approaches by showing a micro example of a product-line (whose programs are 30 lines of code) and isomorphic macro examples (whose programs exceed 30K lines of code), a difference in three orders of magnitude in program size.

Our results provide strong evidence that step-wise refinement scales to large systems, and critical to the scaling of step-wise refinement are multi-dimensional models that compactly describe and specify target product-line members. We begin with a brief description of our prior work.

2 FOP and AHEAD

Feature Oriented Programming (FOP) is the study of feature modularity in product-lines [16]. *AHEAD (Algebraic Hierarchical Equations for Application Design)* is an approach to FOP based on step-wise refinement [2]. The fundamental premise of AHEAD is that programs are constants and refinements are functions that add features to programs. Consider the following constants that represent base 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.

An AHEAD *model* or *domain model* is a set of constants and functions. The set of equations that can be composed from the *units* (i.e., constants or functions) of a model defines a product-line, where each equation corresponds to a member of that product-line.

Since not all features are compatible [7], there are constraints on how units of a model can be legally composed. These constraints, called *design rules*, are not essential to this paper; details are presented in [5].

Implementation. Figure 1a-b show features `K` and `R`. `K` defines a base class `C`; `R` defines a class refinement which adds variable `y` and method `h` to class `C`. In general, a class refinement can add new data members, methods, constructors to a class, as well as extend existing methods and constructors. The composition of `R` and `K`, denoted `R(K)` or `R•K`, is shown in Figure 1c; composition merges the changes of `R` into `K` yielding an updated class definition.

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

(a) `K`

```
refines class C {
  int y;
  void h() {..}
}
```

(b) `R`

```
class C {
  int x;
  int y;
  void g() {..}
  void h() {..}
}
```

(c) `R•K`

Figure 1. Class Definition, Refinement, and Composition

An AHEAD constant is a set of base classes. An AHEAD function is a set of base classes and class refinements. That is, an AHEAD function refines existing classes but can add new classes (which can be subsequently refined) as well. An AHEAD function typically encapsulates a "cross-cut", meaning that it encapsulates fragments (refinements) of multiple classes. Composing AHEAD constants and functions yields packages of fully formed classes [3].

As AHEAD deals with cross-cuts, it is related to Aspect-Oriented Programming. We explain this relationship in Section 5. AHEAD refinements have a long history, originating in collaboration-based designs [17] and their implementations as mixins [21][16] and mixin-layers [19].

3 Multi-Dimensional Models

In this section, we present three examples of multi-dimensional AHEAD models. The first describes a simple product line whose program members are under 30 lines. Our next examples discuss multi-dimensional models whose programs range from 14K LOC to 30K LOC.

3.1 A Micro Example

Consider the AHEAD model `L`, which defines a family 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 with an `ins` (insert) method (Figure 2a).[2]

A refinement of `sgl` is `dbl`, which converts `sgl` into a doubly-linked list (Figure 2b). `dbl` is a "cross-cut" that augments the `node` class with a `prior` pointer, and extends the `list` class with a `last` pointer and refines the `ins` method so that the `last` and `prior` pointers are correctly set.

2. For simplicity, our list programs do not test for invalid inputs.

```
class list {
  node first = null;

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

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

(a) sgl

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

  void ins( node n ){
    if (last == null)
      last = n;
    if (first != null)
      first.prior = n;
    super.ins(n);
    n.prior = null;
  }
}

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

(b) dbl

```
refines class list {
  void del( node n ){
    if (n == first) {
      first = first.next;
      return;
    }
    node prev = first;
    while ((prev != null) &&
          (prev.next != n))
      prev = prev.next;
    if (prev != null)
      prev.next = n.next;
  }
}
```

(c) sgldel

```
refines class list {
  void del( node n ){
    if (n == first)
      first = first.next;
    if (n == last)
      last = last.prior;
    if (n.prior != null)
       n.prior.next =
         n.next;
    if (n.next != null)
       n.next.prior =
         n.prior;
   }
}
```

(d) dbldel

Figure 2. The Units of the L Model

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

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

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

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

Figure 3. Composition dbl•sgl

The composition `dbl•sgl` yields the doubly-linked list program of Figure 3. The code indicated in ***italics*** originates from the `dbl` refinement.

Now suppose we want to enhance our list programs by adding a `del` (delete) method. `sgldel` does exactly this for singly-linked lists: it adds a `del` method to the `list` class (Figure 3c). So a singly-linked list with both `ins` and `del` methods is defined by the composition `sgldel•sgl`.

To create a doubly-linked list that has both insert and delete methods requires a fourth refinement, `dbldel` (see Figure 3d). `dbldel` converts the node deletion algorithm of `sgldel` to a doubly-linked list deletion algorithm.

Using model `L`, we can synthesize the following product-line, which covers all programs that are expressible by `L`:

```
singly_linked_w_ins = sgl                          (1)

doubly_linked_w_ins = dbl •sgl                     (2)

singly_linked_w_ins_and_del = sgldel •sgl          (3)

doubly_linked_w_ins_and_del
       = dbldel •dbl •sgldel •sgl                  (4)
       = dbldel •sgldel •dbl •sgl                  (5)
```

Equations (4) and (5) yield identical programs for inserting and deleting elements in a doubly-linked list. The reason is that the refinements `dbl` and `sgldel` are independent of each other, and thus can be composed in any order.

However, two compositions of `L` are incorrect:

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

Both define programs that are partially implemented. `error1` is a program whose `ins` method works on a doubly-linked list, but whose `del` method works only on a singly-linked list. `error2` is a program whose `ins` method works on a singly-linked list, but whose `del` method works for a doubly-linked list.

The problem in both cases is that if a data structure is extended (i.e., a singly-linked list becomes doubly-linked), then *all* 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 on 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 has refinements (features) that are not truly independent and that groups of refinements (features) must be applied in a lock-step — or all or nothing — manner.

3.1.1 Origami Matrices

Multidimensional models capture the constraint that groups of refinements are applied in a lock-step manner. Model `L` can be abstracted into a 2-dimensional matrix, called an *origami matrix*, where columns represent operations (`delete`, `insert`), and rows represent structure variants (`singleLink`, `doubleLink`). Matrix entries are the refinements of `L` (see Table 1). This matrix can be extended to handle other operations (sort, find) and structure variants (ordered-lists, monitors, etc.).

	`delete`	`insert`
`singleLink`	`sgldel`	`sgl`
`doubleLink`	`dbldel`	`dbl`

Table 1. Origami Matrix for L

To synthesize a particular program, columns of the matrix are composed (or *folded* — hence the name "origami"), where corresponding entries in each row are composed. Table 2 shows the result of folding the `delete` column with the `insert` column.

	`delete • insert`
`singleLink`	`sgldel • sgl`
`doubleLink`	`dbldel • dbl`

Table 2. Column-Composed Origami Matrix

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

The rows 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 `(4)`.

	`delete • insert`
`doubleLink • singleLink`	`dbldel • dbl • sgldel • sgl`

Table 3. A Completely Folded Matrix

By symmetry, instead of folding columns of Table 1, rows can be folded, where corresponding entries in each column are composed (see Table 4).

	`delete`	`insert`
`doubleLink • singleLink`	`dbldel • sgldel`	`dbl • sgl`

Table 4. Row-Composed Matrix

The entry in the `insert` column, `dbl•sgl`, defines program `D` that implements a doubly-linked list with an `ins` method. The entry in the `delete` column, `dbldel•sgldel`, is a refinement of `D` that adds a `del` method. By composing the rows of Table 1, we have synthesized a data structure with a single (`ins`) method, and a refinement that adds a `del` method to this structure. Again, this interpretation holds if we add more rows (structure options) or more columns (operations) to Table 1. Folding the columns of Table 4 yields a 1×1 matrix whose entry is equation `(5)`.

Thus, the constants that can be derived from Table 1, which already exist (e.g., `sgl`) or can be computed by foldings, comprise the product-line of `L`. The inconsistent equations that we encountered earlier, `error1` and `error2`, cannot be derived by folding Table 1.

3.1.2 Perspective

An Origami matrix is a multi-dimensional abstraction of a single-dimensional AHEAD model. In our example, model `L` is abstracted by a pair of orthogonal AHEAD models, each with two units:

```
column  = { insert, delete }              (6)

row     = { singleLink, doubleLink }      (7)
```

The `column` model `(6)` has a lone constant (`insert`) which represents a list program with an `ins` operation. The lone refinement (`delete`) grafts a `del` operation onto the `insert` program. Note: *the `column` model does not capture the details of whether the list program is singly-linked, doubly-linked, key-ordered, etc.; it only reveals operations the list program supports.*

Similarly, the `row` model `(7)` has a lone constant (`singleLink`) which represents a singly-linked list program. The lone refinement (`doubleLink`) transforms the `singleLink` program into a doubly-linked list program. Note: *the `row` model does not capture the details of what operations the list-program has; it only reveals structure variations in lists.*

The folding of a matrix is specified by a set of equations, one per dimension. We call these *dimensional equations*. Each dimensional equation tells us how to fold (or collapse) a dimension to a single unit.[3] For example, a singly-linked list with both an insert and deletion operation is defined by a pair of dimensional equations:

```
col_eqn = delete • insert                 (8)

row_eqn = singleLink                      (9)
```

The `col_eqn` folds the `delete` column with the `insert` column; the `row_eqn` discards all but the `singleLink` row. The result of this folding is equation `(3)`. *In effect, each dimensional equation specifies a different "projection" of the target program. The intersection of all these "projections"*

3. The rules by which units are composed along each dimension are specified by design rules [5].

uniquely identifies the target program that is to be synthesized.

Origami or multi-dimensional models provides a fundamental form of scalability to AHEAD and FOP. Given an *n*-dimensional matrix where each dimension has *k* units, a program is specified by *n* dimensional equations, each referencing up to *k* units. Thus a program specification in Origami has up to *O(kn)* terms. However, the number of units that are folded together to produce an equation for the target program, as expressed in a 1-dimensional model, is $O(k^n)$. *In short, multi-dimensional models provide compact specifications for potentially very complex systems.*

3.2 The Bali Matrix

The AHEAD model of Section 2 is supported by a sophisticated set of tools. The *AHEAD tool suite (ATS)* includes a set of compiler-compiler tools for synthesizing families of language dialects [2]. ATS tools have been bootstrapped, and are written in an extended dialect of Java called Jak (short for Jakarta). ATS provides three tools to extend Jak (or any language for that matter): `balicomposer`, `bali2javacc`, and `bali2jak`. Base grammars and their refinements are expressed as annotated BNF grammars called *Bali grammars*. `balicomposer` composes Bali grammars, `bali2javacc` translates a Bali grammar to a `javacc` grammar, and `bali2jak` generates a customized set of Jak classes for defining semantic actions for parsers.

Originally, each tool was defined by a hand-crafted equation using the units of an AHEAD model `B`. We now use a 2-dimensional Origami matrix to synthesize these equations. The dimensional models are listed below:

```
row = { core, codegen, require, tool }
column = { base, b2jak, b2jcc, bc }
```

The `row` model defines a layered design shared by all Bali tools. `core` defines a common infrastructure, `codegen` grafts on code-generation capabilities, `requires` adds optional import statements to Bali grammars, and `tool` adds tool-specific processing. There are only two legal equations that can be synthesized from the `row` model currently — one with require, the other without:

```
wRequire = tool • require • codegen • core     (10)

woRequire = tool • codegen • core              (11)
```

To understand the `column` model, observe that all Bali tools take a Bali grammar file as input and produce different outputs. All tools share a common parser and lexical analyzer, but differ in the semantic actions performed on parse trees. These relationships are captured by the `column` model. The `base` unit encapsulates the common parser and lexical analyzer; the remaining units encapsulate the semantic actions for each tool (`b2jak` encapsulates the actions of the `bali2jak` tool, `b2jcc` encapsulates the actions of the `bali2javacc` tool, etc.). Thus, there are three legal equations for the `column` model, one equation for each tool:

	base	b2jak	b2jcc	bc
core	parser			
codegen		cgen	cgen	
require	reqGram	reqb2jak	reqb2jcc	reqComp
tool		b2jktool	b2jcctool	composr

Table 5. Bali Matrix

```
balicomposer = bc •base                 (12)

bali2javacc = b2jcc •base               (13)

bali2jak = b2jak •base                  (14)
```

The Origami matrix that relates these models is Table 5.

Each Bali tool is specified by a pair of dimensional equations: either `wRequire` or `woRequire` and the equation that defines the tool itself (`(12)`, `(13)`, `(14)`). The equation that is synthesized by folding Table 5 for `balicomposer` with requires is:

```
balicomposer =
        composr •reqComp •reqGram •parser
```

Although the ideas of matrix folding are identical to that of our micro example of Section 3.1, the size of the programs that are synthesized are very different. Each list program is barely 30 lines long; each Bali tool is about 14K LOC. The product-line we discuss in the next section has tools that commonly exceed 30K LOC.

3.3 The Jak Matrix

Our first use of Origami was in the synthesis of Jak tools [3]. ATS has tools to compose Jak files, to decompose previously composed Jak files, to translate Jak files to Java files, to pretty-print Jak files, and to document Jak files (ala `javadoc`). A 3-dimensional (8×6×8) matrix is used to capture their common design. The shape of this matrix is displayed in Figure 4a; for the most part, it is empty except for the `Frontal` matrix and the `Horizontal` matrix. The `Frontal` matrix is just like the Bali matrix: columns are tool features (e.g., common parser and semantic action packages for each tool) and rows correspond to language features. That is, the lone constant represents the Java language and the refinements are extensions to Java (e.g., state machines, refinement declarations). The `Horizontal` matrix captures tool feature-language feature interactions, and has exactly the same dimensions as the `Frontal` matrix.

To build a particular tool, three dimensional equations are needed: there is a row equation (which specifies the variant of Java to produce) and a column equation (to pair a parser shared by all tools with the package that defines tool-specific semantic actions). The third equation folds the `Horizontal` matrix into a single row of the `Frontal` matrix.

A Jak tool is synthesized as shown in Figure 4b-e. The `Horizontal` matrix is folded into a single row of the `Frontal` matrix (Figure 4c); the `Frontal` matrix rows are folded (Figure 4d), and the columns are folded (Figure 4e), to yield a 1×1 matrix which contains the tool equation [3]. We sketch its salient characteristics as we will show an alternative encoding/interpretation for it in Section 4.4.

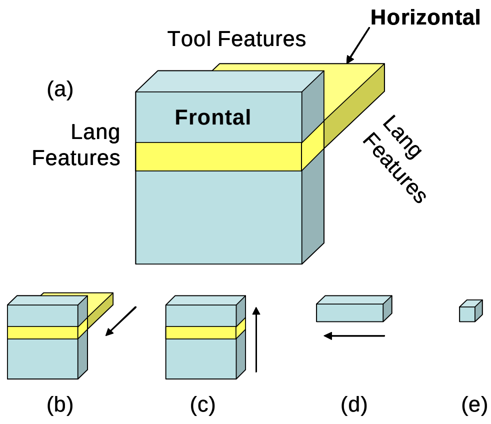

Figure 4. Jak Matrix and its Foldings

3.4 Future Directions

We are using Origami to build suites of Jak and Bali tools and envision additional matrices to synthesize tools for other AHEAD program representations. An example is design rule files. A *design rule file* specifies the preconditions and postconditions for unit usage; it is written in a special predicate language [5]. We now have a tool, called `drc`, which composes design rule files and reports composition errors. We expect that additional tools for design rule files, such as a pretty-printer and a `javadoc`-like utility, will be created. To ensure that the designs of these tools are consistent, we would synthesize them from an Origami matrix.

4 Reinterpreting Origami

Given the examples and directions of the previous section, it is clear that Origami and MDSOC should be a fundamental part of the AHEAD model. But an AHEAD model is simply a set, *not* a matrix, of units. So how should matrices be integrated into AHEAD?

We wrestled with these and related questions for some time, because introducing an ad hoc concept into AHEAD would unnecessarily complicate its construction and maintenance for years to come. It dawned on us that Origami is part of a much bigger picture governed by AHEAD, and requires no fundamentally new concepts, except a new representation of object models.

4.1 The Principle Of Uniformity

Software architects routinely use many different representations, including code, to specify a program. Non-code artifacts such as UML documents, performance models (e.g., Mathematica equations), makefiles, etc. are common.

In general, a program `P` will have a set of distinct, but interrelated representations $\{P_1 \ldots P_n\}$. When a new feature `F` is added to a program (e.g., $F \bullet P$), multiple representations are affected. Not only is the code representation of the program changed (because `F` must be implemented), so too will changes occur in its UML representation (to document `F`'s changes), its performance model (to account for `F`'s performance impact), its makefiles (to build `F`), and so on. Thus, a refinement `F` will have a set of distinct but interrelated "sub" refinements $\{F_1 \ldots F_n\}$, where F_i defines how the *i*th representation of a program is refined. It follows that the composition $F \bullet P = \{F_1 \bullet P_1 \ldots F_n \bullet P_n\}$ defines the set of all representations of the `F`-refined program `P` [2].

We know how code artifacts can be refined — we have shown examples in previous sections. But how are non-code artifacts refined? AHEAD is based on a general principle that governs how all artifacts, both code and non-code, can be refined. This is the Principle of Uniformity [2].

The *Principle of Uniformity* states (1) that all artifacts — code and non-code — can be given a class structure and (2) that this structure can be refined. Thus, the refinement of code artifacts is essentially no different than the refinement of non-code artifacts; although obviously differing in implementation details, the ideas are fundamentally the same.

This principle was used to great effect in LISP and Smalltalk, which were both programming languages *and environments*. Namely, just as objects are instances of classes, files are instances of file types. Methods on files correspond to tools. For example, methods that can be invoked on Java files are `javac` (compile), `javadoc` (document), `emacs` (edit), and so on. Methods that can be invoked on HTML files are `browser` (view), `Acrobat`™ (to convert to PDF files), `FrontPage`™ (edit), etc. Even directories, which define sets of file instances, have methods: open, explore, search, rename, etc.

Operating systems such as Windows™ provide an object-based view of their file systems to users. This view is not object-oriented as there is no inheritance hierarchy that relates different file types and their instances. *But there could be.*

Consider a class diagram that captures the relationships among AHEAD file types and tools. A file method (i.e., operation on a file) either modifies that file (e.g., `reform` — which transforms an unformatted Jak file into a nicely formatted Jak file) or transforms it from one type to another (e.g., `javac` maps Java files to class files).

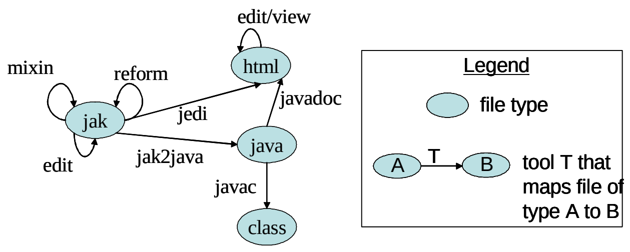

Figure 5. An Object Model of ATS File Types and Tools

Thus, part (1) of the Principle of Uniformity tells us that ATS implements an object model shown in Figure 5. Part (2) of the Principle tells us that we can create variations in this tool suite by refinements. A refinement of an object model is simple: it can add new methods/tools to existing classes, it can refine existing methods/tools, and can add new types. Thus, we can (and have!) built variants of ATS with and without design rule file types; we can (and have!) built variants of ATS with and without `reform` methods/tools, and so on.

In a bootstrapping maneuver, the AHEAD Tool Suite and its variations can be expressed as an AHEAD model/product-line. That is, base ATS file types are defined by an AHEAD constant, and refinements of this base are AHEAD functions. We will see Section 4.3 how Origami matrices are related to this model.

4.2 Representation of File Types

Let us define a notation to represent a file type and its methods in AHEAD. The notation we use is illustrated below: a file type is given a name (`fname`) and each method/tool is specified by an equation which defines the set of units that are composed to synthesize that method/tool. `tool1`, for example, is implemented by the composition of units (`a`—`f`). The specification below defines a file type called `fname` and three tools that can be invoked on `fname` instances.

```
type fname {
   tool1 = a •b •c •d •e •f
   tool2 = x •y •z •d •e •f
   tool3 = q •r •s •d •e •f
}                                        (15)
```

Tool equations have common subexpressions that can be factored out as helper methods (which would correspond to shared packages that are referenced/imported by tools), as shown below. Note that `(15)` and `(16)` are semantically equivalent.

```
type fname {
   common = d •e •f
   tool1 = a •b •c •common
   tool2 = x •y •z •common
   tool3 = q •r •s •common
}                                        (16)
```

Given this class-like representation, we can define a refinement of this specification in a straightforward manner. The specification below refines `fname` by extending the `common`, `tool1`, and `tool2` methods, and introducing a fourth tool, `tool4`.

```
refines type fname {
   common = super.common •g
   tool1 = m •super.tool1
   tool2 = super.tool2 •n
   tool4 = p •q •common
}                                        (17)
```

Method/tool refinement is specified via the `super.t` construct; it means substitute the previous equational definition of tool `t`. Thus, if tool `t` is defined by:

```
t = x •y •z
```

and a refinement of `t` is:

```
t = super.t •q •r •s
```

Their composition yields:

```
t = x •y •z •q •r •s
```

The composition of `(16)` and `(17)` yields the specification:

```
type fname {
   common = d •e •f •g
   tool1 = m •a •b •c •common
   tool2 = x •y •z •common •n
   tool3 = q •r •s •common
   tool4 = p •q •common
}                                        (18)
```

The identity function is denoted by `id`. Any unit `x` composed with `id` is `x`:

```
x = x •id                                (19)
```

Given the above, we can represent a base object model — a set of core types and core methods/tools — as an AHEAD constant. This constant would consist of one or more specifications of the form illustrated by `(16)`. A refinement of an object model would be an AHEAD function. It would consist of zero or more specifications of the form `(16)` and `(17)`, that is, existing types could be refined and new ones added. An AHEAD model of ATS would consist of a set of these constants and functions.

4.3 Origami Matrices

An Origami matrix defines multidimensional relationships among refinements that are used to synthesize a suite of tools for a particular file type. All of the matrices that we have seen to date have the following general structure: rows represent “structural” features and columns represent optional tools (e.g., methods and helper methods). Thus, a straightforward mapping of an Origami matrix to an AHEAD model is to represent each row of a matrix as a unit (constant or function) in the model. For example, a model that underlines the Bali matrix is:

```
Bali =
  { core, codegen, require, tool, toolset }(20)
```

where `code`, `codegen`, etc. are rows of the matrix. (We will explain the `toolset` unit of `Bali` shortly). The file type representations of the `core` and `codegen` rows, for example, are shown in Figure 6. The listed methods are helpers or refinements of helpers.

```
(a)       type BaliFile {
core         base = parser
             b2jak = id
             b2jcc = id
             bc = id
          }

(b)       refines type BaliFile {
codegen      b2jak = cgen • super.b2jak
             b2jcc = cgen • super.b2jcc
          }
```

Figure 6. Representation of Matrix Rows

We need one additional file type specification that defines an equation — a column folding — for every tool. The name of this specification is `toolset`:

```
refines BaliFile {
   bali2jak     = b2jak • base
   bali2javacc  = b2jcc • base
   baliComposer = bc • base
}
```

That is, `toolset` encodes equations `(12)-(14)`. When a new Bali tool is added, the `toolset` file is updated with a new method (the tool’s equation).

Given model `(20)`, we can synthesize the two variants of the Bali tool suite by:

```
noRequires = toolset • tool • codegen • core
wRequires  = toolset • tool • requires
                  • codegen • core
```

The resulting file type specification, `noRequires` or `wRequires`, has an equation for each helper method and tool method. When a tool equation is evaluated, the referenced units are composed and a tool is synthesized. This is how we are synthesizing Bali tools.

4.4 Additional Dimensions

The Jak matrix is more complicated because feature-feature interactions require a third dimension. We can still use file type representations, but with a bit more sophistication. To understand our solution, consider a common situation that arises when multiple classes are simultaneously refined.

Figure 7a shows types/classes `F` and `H`, each with one method. Figure 7b shows these classes after they have been refined. Note that method `m` of class `F` calls method `n` of class `H`. Figure 7c shows the result of one more refinement. When method `m` of class `F` of Figure 7c is evaluated, it executes statement `a`; dispatches to method `n` of `H` where statements `x`, `y`, `z` are executed; and returns to execute statement `c`.

```
type F {         type F {           type F {
   m = a            m = a•H.n          m = a•H.n•c
}                }                  }

type H {         type H {           type H {
   n = x            n = x•y            n = x•y•z
}                }                  }
    (a)              (b)                (c)
```

Figure 7. Refinement of a Pair of File Type Specs

This is how we encode and evaluate file type specifications for the Jak matrix. `F` and its refinements correspond to rows of the `Frontal` matrix. `H` and its refinements correspond to rows of the `Horizontal` matrix. By refining `F` and `H` simultaneously, in effect we are simultaneously folding both the `Frontal` and `Horizontal` matrices in lock-step. The resulting equations are exactly the same had we refined `H` separately from `F` (that is, by folded the `Horizontal` matrix first and then the `Frontal` matrix next).

This example suggests that additional dimensionality in Origami matrices does not require anything extra in AHEAD. We can map extra dimensionality to extra feature sets, and the refinement of these sets, all within the AHEAD framework. Doing so, we have conceptually simplified the generation of Jak tools and have also simplified the matrix folding algorithms and their implementations.

4.5 Reinterpreting MDSOC

So where does MDSOC fit into AHEAD? The above discussions suggests that it simply disappears into AHEAD formalisms. In reality, it does not.

An Origami matrix `D` (or MDSOC model) is a multi-dimensional *abstraction* or *view* of a 1-dimensional AHEAD model `M`. Using the constants and functions of `M`, huge numbers of equations — many of them nonsensical — can be created, each representing different tools or variants of a particular tool. From our experience, it is more difficult to interpret equations from `M`, more difficult to write design rules for `M`, more difficult to explain the units of `M`, and so on, simply because `M` is so close to an implementation.

On the other hand, a multi-dimensional model `D` is easier to understand. Because it is an abstraction of `M`, there are many fewer units in `D`. Each is easier to explain, compositions of units are easier to interpret, and writing design rules is simplified.

The set of all correct equations that can be derived from model `D` is a subset of all correct equations that can be derived from model `M`. If the abstraction of `M` to `D` is conceived properly, all interesting equations that are synthesizable by `M` are synthesizable by `D`; correct equations that are not synthesizable by `D` are either uninteresting — we don't care about these compositions — or can be inferred (via algebraic rewrites) from equations that can be derived.

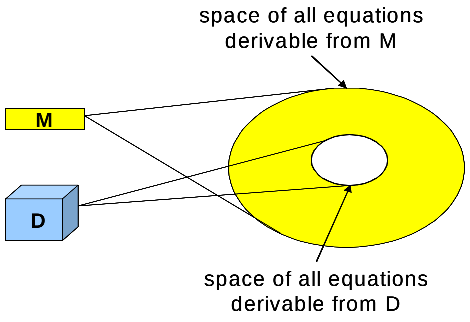
space of all equations derivable from M
M
D
space of all equations derivable from D

Figure 8. Relationships among 1-D models and n-D Models

Thus our interpretation of MDSOC is a design and abstraction process by which a low-level model `M` is abstracted into a higher-dimensional model `D`. Both `D` and `M` are AHEAD models, but `D` is a much simpler model that is used to synthesize equations for `M`.

5 Related Work

As mentioned earlier, AHEAD refinements encapsulate cross-cuts, that is, fragments of classes. The idea of cross-cuts was popularized by Aspect-Oriented Programming [13][12], so it is interesting to see how aspects of AspectJ relate to AHEAD refinements. There are two basic differences. First, the concept of refinement in AHEAD (and its predecessor GenVoca) is virtually identical to that of extending object-oriented frameworks. Adding a feature to an OO framework requires certain methods and classes to be extended. AHEAD takes this idea to its logical conclusion: instead of having two different levels of abstraction (e.g., the abstract classes and their concrete class extensions), AHEAD allows arbitrary numbers of levels, where each level implements a particular feature or refinement [4].

Second, the starting points for AHEAD and AOP/AspectJ are different: product-lines are the consequence of pre-planned designs (so refinements are designed to be composable); this is not a part of the standard AOP/AspectJ paradigm. Stated another way, the novelty and power of AspectJ is in quantification [1]. Quantification is the specification of where advice is to be inserted (or the locations at which refinements are applied). The use of quantification in AHEAD is no different than that used in traditional OO designs.

An important consequence of the use of pre-planned designs is simpler tools. Consider the following assignment statement, where `a`, `b`, `c` denote method calls and • denotes method/function composition:

$$X = a \bullet b \bullet c \qquad (21)$$

In AspectJ, one can add advice to method calls. Equation `(22)` shows the result of adding "after" advice to each term in `(21)`, and `(23)` shows the result of adding another layer of after advice to each term in `(22)`:

$$X = a_{after1} \bullet a \bullet b_{after1} \bullet b \bullet c_{after1} \bullet c \qquad (22)$$

$$X = a_{after2} \bullet a_{after1} \bullet a \bullet b_{after2} \bullet b_{after1} \bullet b \bullet c_{after2} \bullet c_{after1} \bullet c \qquad (23)$$

Equations `(21)-(23)` can be derived easily by folding the rows Origami matrix of Table 6, where the first row defines the initial terms and subsequent rows define particular after advice. By folding rows and then columns, equations `(21)-(23)` follow. Before and around advice can also be encoded in our notation.

	C	B	A
base	c	b	a
1stAdvice	c_{after1}	b_{after1}	a_{after1}
2ndAdvice	c_{after2}	b_{after2}	a_{after2}

Table 6. Origami Matrix Emulating Before Methods

We originally folded matrices using shell scripts; currently we use `ant` scripts [23]. Thus the tools that we need to "annotate" or "insert advice" into designs are very simple, an advantage of our approach. Another advantage is representation generality. By using equations, we can define and refine both source code and non-code representations of tools etc. with exactly the same formalism.

A predecessor to MDSOC is the work by Harrison and Ossher on subjectivity [10], which embodies a fundamental observation about software design: an object does not have a single interface, but rather has a large number of interfaces. The interface that is given to an object is *subjective*, that is, one that is specific to the task at hand. As an example, consider a book. Obvious attributes (with their accompanying set and get methods) are name, title, and author. But these attributes/methods are useful only for library-like applications; they are not useful for warehouse applications (where attributes of physical size are critical — name, title, author are useless), or for publishing applications (how much ink and paper are needed?). Subjectivity is a fundamental part of Origami. An Origami matrix says all tools are refined when feature `F` is added, but the meaning of `F` is specific to a tool. There is no single definition of refinement `F`, but rather different interpretations/implementations specific to the tool under consideration.

6 Conclusions

Modularity has multiple goals: to encapsulate functionality that can be reused, to hide implementation details, and to offer more compact ways of specifying systems out of larger parts. The history of modularity shows a progression of larger abstractions (e.g., functions, suites of functions (e.g., classes), suites of classes (packages or components)). Most recently, the concept of "fortresses" has been introduced to talk about the encapsulation of enterprise software architectures [18]. Each new abstraction is quite different than the ones before it; each has different properties, capabilities, scale, and applications. Concepts that apply at one level of modularization may not apply at other levels.

Step-wise refinement offers an alternative to this trend. We have shown how equational specifications can be used to define product-lines in the small (e.g., programs of 30 lines) and in the large (e.g., programs of 30K lines), a difference of three orders of magnitude in program size. We have shown how a small set of ideas can be uniformly applied at different levels of abstraction. Further, we believe that equational representations scale to systems of arbitrary size and complexity.

Equational representations alone are not enough to achieve scalability. Combining them into multi-dimensional models, a form of MDSOC, provides much smaller descriptions of complex programs. In this paper, we showed that programs whose specifications might be $O(k^n)$ terms in length had short and more understandable specifications of length $O(kn)$ when multi-dimensional models are used.

As the emphasis on automation in software development increases, we believe that equational representations of programs will play an increasing role in automated program synthesis. Our work provides strong evidence that equational representations based on step-wise refinement scales to very large systems.

Acknowledgements. We thank Roberto Lopez-Herrejon for his helpful comments on clarifying drafts of this paper.

7 References

[1] AspectJ. Programming Guide. `http://aspectj.org/doc/proguide`

[2] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement", *ICSE 2003*.

[3] D. Batory, R. Lopez-Herrejon, J.P. Martin, "Generating Product-Lines of Product-Families", 2002 *Automated Software Engineering Conference*. Revised paper submitted to journal publication.

[4] D. Batory, R. Cardone, and Y. Smaragdakis, "Object-Oriented Frameworks and Product-Lines". *1st Software Product-Line Conference*, Denver, Colorado, August 1999.

[5] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, Feb. 1997, 67-82.

[6] I. Baxter, "Design Maintenance Systems", *CACM*, April 1992.

[7] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report", GCSE/SAIG 2002.

[8] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[9] M. Griss, "Implementing Product-Line Features by Composing Component Aspects", *First International Software Product-Line Conference*, Denver, August 2000.

[10] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-427.

[11] K.C. Kang, et al., Feature-Oriented Domain Analysis Feasibility Study, SEI 1990.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.

[13] G Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, W.G. Griswold. "An overview of AspectJ". *ECOOP 2001*.

[14] H. Ossher and P. Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM 44(10): 43-50, *October 2001*.

[15] H. Ossher and P. Tarr, "Multi-dimensional separation of concerns and the Hyperspace approach." In *Software Architectures and Component Technology* (M. Aksit, ed.), 293-323, Kluwer, 2002.

[16] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects", ECOOP 1997.

[17] T. Reenskaug, et al., "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", *Jour. OO Programming*, 5(6): October 1992, 27-41.

[18] R. Sessions, "Software Fortresses: Modeling Enterprise Architectures", Addison-Wesley, 2002.

[19] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", *ACM TOSEM*. March 2002.

[20] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*.

[21] M. Van Hilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs", *OOPSLA 1996*, 359-369.

[22] D. Weiss and C.T.R. Lai, *Software Product-Line Engineering*. Addison-Wesley, 1999.

[23] The Apache Ant Project. `http://ant.apache.org/`