Frame-Based Code Generation of Multi-Dimensional Feature Sets or Origami meets Frames

Roberto E. Lopez-Herrejon and Jacob Neal Sarvela
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{rlopez, sarvela}@cs.utexas.edu

Abstract. Software product lines are defined in terms of feature sets that can be orthogonally arranged, according to different design criteria, to constitute dimensions in a design space. When feature sets are well-structured, it is possible to use *Origami*, a feature-oriented programming technique, to guide feature modelling and composition. However, a considerable manual effort still remains to maintain the consistency of *multi-dimensional feature sets*, as feature implementations are parametrically interdependent. In this paper, we present a code-generation technique that integrates *frame technology* and Origami to address this problem. Our motivating example is the *expression problem*, viewed as an instance of a two-dimensional feature set.

1 Introduction

Software product lines are defined in terms of feature sets. Many techniques and methodologies have been proposed to build product-line applications with different feature models, representations, and generation techniques (e.g. [8]).

Sometimes features sets can be orthogonally arranged, according to different design criteria, to constitute dimensions in a design space. We have developed AHEAD [4][5], a methodology that models multi-dimensional feature sets as multi-dimensional matrices, called *Origami* matrices. A matrix entry is a module that implements the "semantic intersection" of orthogonal features. We have used Origami to synthesize large systems (in excess of 250K Java LOC) in an algebraic, incremental, and scalable fashion [5]. Although Origami greatly reduces the effort to build systems, a considerable manual effort still remains to maintain the consistency of multi-dimensional feature sets, as their implementations are parametrically interdependent. In this paper, we present a technique that uses frame technology to capture such commonality with the goal of generating portions of Origami matrix entries, therefore reducing the effort to maintain matrix consistency. Doing so reveals a new class of design support tools for program development.

We illustrate our technique on a simple, yet fundamental, software design problem: That of extending an application with new operations and new data types. To build on common ground, we borrow an instance of this problem from programming languages literature where it is referred to as the *expression problem* [18].

2 Running Example: The Expression Problem

Product-line methodologies use feature models to support *extension* (adding new features to product-line applications), and *selection* (specifying feature combinations to generate concrete member products). A typical extension scenario adds features to support a mix of new operations and data types. This scenario has been extensively studied within the context of programming languages design, under the name of the *expression problem* [18][7]. In this context, the primary focus is to achieve data type and operator extensibility in a type-safe manner, without resorting to code modification or repetition, and avoiding run-time type errors [17][9]. Though important issues by themselves, our focus is different as we concentrate on the *design and synthesis* aspects of the expression problem: How can a set of features be modelled in an extensible manner, so that adding new data types and operations, and generating software products with them are simple tasks?

As our concrete example, we adapt Torgersen's expression problem [17]. The goal is to define data types to represent expressions of the following language:

```
Exp  :: = Add | Lit
Add  :: = Exp "+" Exp
Lit  :: = <non-negative integers>
```

Associated with this grammar is an operation called `ToString` that computes the string value of an expression. For example, the expression 2+3 can be represented as a three-node tree with an `Add` data type node as the root and two `Lit` data type nodes as leaves. The operation `ToString`, applied to this tree, produces the string "`2+3`".

This application can be extended by adding new data types. To support negation of expressions requires the following grammar extension:

```
Exp  :: = ... | Neg
Neg  :: = "-" Exp
```

where `...` denotes the previous right-hand side of `Exp`. The application can also be extended by adding new operations, such as `Eval`, a function that evaluates expressions and returns their numeric value. Applying the operation `Eval` to the tree of expression 2+3 yields 5 as result.

A natural representation of the expression problem is a two-dimensional matrix [18][7][9]. Consider the original grammar with expression data types `Lit` and `Add`. Supporting operation `ToString` for them can be represented as a 2×1 matrix in Figure 1a where the horizontal dimension specifies the set of operations, while the vertical dimension specifies the set of data types that implement them. Each matrix entry defines a module whose code implements the operation given by the column for the data type given by the row.[1]

1. The `Exp` type actually is present in this example; it is implicitly contained within the `Lit` row. We will make it explicit in a row later.

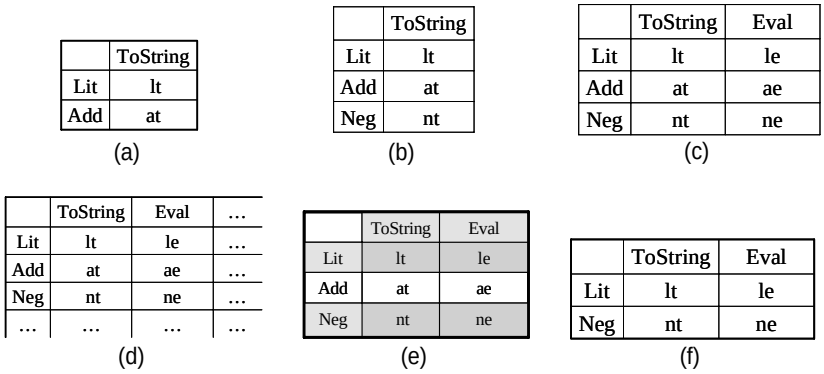

Figure 1. Matrix representation of Expression Problem

As a naming convention throughout the paper, we identify matrix entries by using the first letters of the row and the column. For example, the entry at the intersection of row `Add` and column `ToString` is named `at`.

When a new data type is added, the change is reflected by adding a new row to the matrix and implementing the operations specified by the columns. When data types are extended with `Neg`, for example, the matrix is modified as shown in Figure 1b. Furthermore, adding a new operation modifies the matrix by adding a new column and implementing the operation on the data types specified by the rows. The result of adding operation `Eval` is shown in Figure 1c.

This arrangement of rows and columns, illustrated in Figure 1d, is the key to two-dimensional extensibility. On one hand, adding a new data type entails creating a new row and filling its corresponding column entries while, on the other hand, adding a new operation implies creating a new column and filling the corresponding rows. Feature sets that can be arranged as matrices of two or more dimensions are *multi-dimensional*.

Within a product-line, a software designer should be able to select exactly those features required for an application. In our example, this means specifying desired rows and columns in a feature matrix. Consider an application that contains `ToString` and `Eval` operations for `Lit` and `Neg` data types. The selection of these features forms regions in the matrix, as illustrated in Figure 1e. These regions are then projected, as shown in Figure 1f, to form a matrix for the target application. The question now is, how can this matrix representation of the expression problem be translated into a program?

3 Feature Oriented Programming and AHEAD

Feature Oriented Programming (FOP) is the study of feature modularity in product-lines [15]. *AHEAD (Algebraic Hierarchical Equations for Application Design)* is an approach to FOP that is based on step-wise refinement [6]. AHEAD arranges sets of orthogonal features in structures called *Origami* matrices [4], which makes this approach a natural fit for the expression problem.

The central idea of AHEAD is to divide features in two categories:

- **Constants** that are base programs. For example,

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

- **Refinements** are *functions* which receive a program as input, add features to it, and produce a new program as output. For example,

```
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
```

where • denotes function composition. 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 all equations that can be composed from the elements of the model defines a *product-line*, where each equation defines one of its members.

Implementation. A *constant feature* is a set of Java classes and interfaces. An example is feature `C` in Figure 2a, which encapsulates interface `I` and class `A` that implements `I`. The set notation in Figure 2 denotes encapsulation.

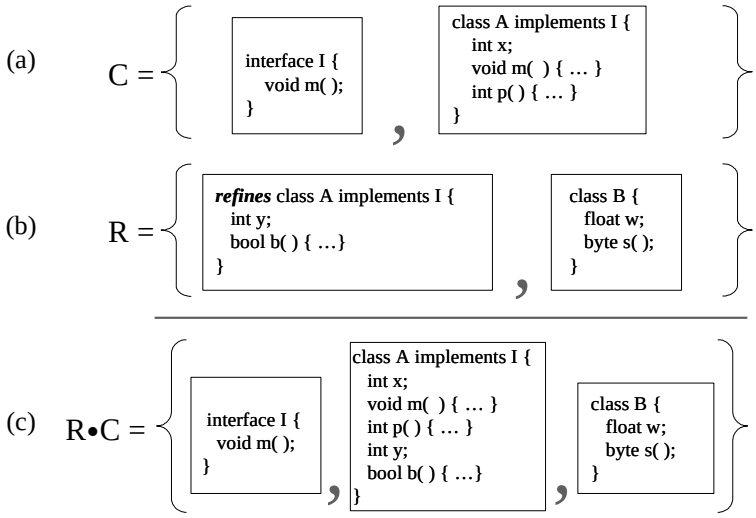

Figure 2. Constants, Functions, and Composition

A *function feature* is a set of refinements of classes and interfaces that can add fields, methods, constructors, method refinements and constructor refinements. A *function feature* can also add new classes and interfaces that can be subsequently refined or extended. An example is feature `R` in Figure 2b, which contains a refinement to class `A` that adds field `y` and method `b`, and also adds class `B`. AHEAD tools use a language,

called *Jak* [4][5], that is a superset of Java. One of its keywords is the modifier *refines* which distinguishes normal classes and interfaces from their refinements.

The composition of `R` and `C`, denoted by `R • C`, is Figure 2c. `R•C` yields a new set formed with interface `I`, class `A` with members from the base class (field `x`, methods `m` and `p`) plus `A`'s refinement members (field `y` and method `b`), and class `B`.

3.1 Implementing the Expression Problem with AHEAD

In Section 2 we saw how the expression problem can be represented as a two-dimensional matrix and how our concrete example can be depicted as shown in Figure 1c. To represent multi-dimensional designs, AHEAD uses *Origami* matrices [4] whose entries are AHEAD constants and functions. Figure 3 illustrates an implementation of the expression problem in an Origami matrix. Note that the structure, rows and columns, of the solution is the same as that of Figure 1c.

Consider entry `lt`, in row `Lit` and column `ToString`, in Figure 3. This entry is a constant feature with two members: Interface `Exp` that declares method `toString()`, and class `Lit` with a `value` field, a constructor, and method `toString()` to implement `Exp`. Extending `Lit` with `Eval` operation requires: refining interface `Exp` to add the signature of method `eval()`, and refining class `Lit` to add its implementation. These refinements are implemented by the function feature shown in entry `le` of Figure 3.

	ToString	**Eval**

Lit	**lt**
	interface Exp { ***String toString( ) ;*** }
	class Lit implements Exp {
	public int value;
	Lit (int v) { value = v ; }
	String toString( ) { return String.valueOf(value); }
	}
		le
		refines interface Exp {
		int eval ( ) ;
		}
		refines class Lit
		implements Exp {
		int eval( ) { return value; }
		}

Add	**at**
	class Add implements Exp {
	public Exp left, right ;
	Add (Exp l, Exp r) { left = l ; right = r ; }
	String toString( ) { return left.toString( ) + "+" +
	right.toString( ); }
	}
		ae
		refines class Add
		implements Exp {
		int eval( ) {
		return left.eval ( ) +
		right.eval ( );
		}
		}

Neg	**nt**
	class Neg implements Exp {
	public Exp expression;
	Neg (Exp e) { expression = e ; }
	String toString( ) { return "-(" + expression.toString( ) + ")" ;}
	}
		ne
		refines class Neg
		implements Exp {
		int eval( ) {
		return -expression.eval( ) ;
		}
		}

Figure 3. A solution to the Expression Problem with AHEAD

Now consider entry `at`, in row `Add` and column `ToString`, in the same figure. This entry is a feature that defines class `Add`, with two operand expressions, a constructor, and method `toString()` to implement `Exp`. Extending `Add` with the `eval()` operation is accomplished by feature `ae`. The data type and `Eval` extension of `Neg` is implemented in a similar way to that of `Add` as illustrated in row `Neg` of Figure 3.

The question now is: How are the selected features of an application composed? To illustrate how this process works, recall the problem of creating an application that supports operations `ToString` and `Eval` for `Lit` and `Neg` data types, as described in Section 2.

We start with the matrix in Figure 4a that shows the regions formed by the selected application features. The first step is to project those regions to a new matrix as depicted in Figure 4b. This Origami matrix, by analogy to the similarly named Japanese art of paper folding, is composed by *folding* along rows and columns. For example, Figure 4c shows the result of folding columns `Eval` and `ToString` to form a composite column `Eval•ToString`. When columns are folded, corresponding entries in each row are composed. Thus, the expression `le • lt` is synthesized for the `Lit` row and expression `ne • nt` is produced for the `Neg` row in the composite column. Once columns are folded, we compose the corresponding rows to form a composite row (`Neg•Lit` in Figure 4d). Alternatively, we could have composed rows first, and then the resulting columns. Although different expressions would result, these expressions would be semantically equivalent [6][1].

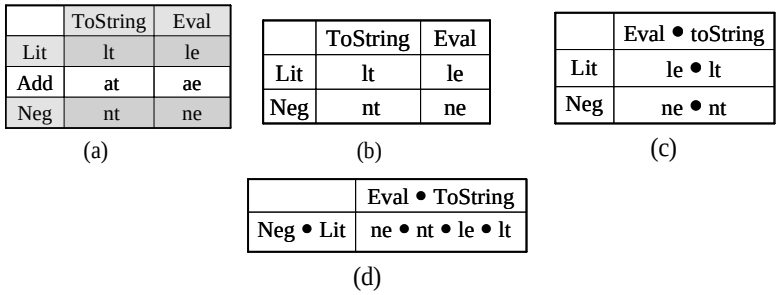

Figure 4. Matrix folding and composition

Composition ends with a 1×1 matrix (Figure 4d). The entry defines the AHEAD expression that can synthesize our application:

```
LitNegApp = ne • nt • le • lt
```

Note that folding column and row features must follow an order that satisfies the domain design rules of the product family. For example, the folding of columns in order `ToString•Eval` is not valid since features in column `Eval` refine those in column `ToString`. How design rules are expressed in AHEAD is not relevant for this paper, interested readers are referred to [3]. The details of matrix implementation, composition, folding, and generation are also not critical; for further details see [6][1].

4 Framing Origami

In the previous section, we saw a direct implementation of the expression problem with Origami. There, program refinements consist of adding a new method definition to an interface and its implementation to a class. However, it is often the case that more complex class and interface collaborations exist between features. As an example, we present a second implementation of the expression problem and use this implementation later as the basis to illustrate our generation technique.

4.1 A Visitor Implementation

Design patterns are well-known programming protocols that capture complex collaborations between objects of different classes and interfaces. The *Visitor* design pattern specifies an operation on a family of data types by encapsulating the operation's implementation within a separate *visitor class* [10]. The connection between the visitor class and the data type classes is specified by defining two sets of methods:

- The visitor class of each operation defines *visit* methods, one for each data type. Each method is a type-specific implementation of the visitor's operation.
- Each visited data type implements an *accept* method for a type of visitor class. These methods call the corresponding *visit* methods of the accepted visitor object, usually passing a self-reference as an argument.

Figure 5 illustrates an implementation of this approach for classes `Lit` and `Add` with operation `ToString`. It works as follows:

- Class `ToStringV` implements operation `ToString`. It contains *visit* methods for classes `Lit` and `Add` that define the operation for the two data types.
- Classes `Lit` and `Add` each declare an *accept* method with a parameter of type `ToStringV`. These methods respectively call *visitLit* and *visitAdd*, with a self-reference (`this`).

Previous work [17][18][12] has shown that visitors do not solve the expression problem because they have the following failings:

- Extending the set of data types requires the modification of the visitor classes to add methods to visit the new data types.
- Supporting new operations with different parameter lists or return types (like `Eval` that returns an `int` value), requires the modification of the data types to add *accept* methods tailored for those operations.

The value of Origami is evident because it allows refinement of existing visitor classes and data types with *visit* and *accept* methods, and it supports the incremental inclusion of new visitors and data types. The detailed Origami implementation that we present now uses a variation of the visitor pattern, where the *visit* methods call a method in the data types to perform the operation instead of performing it themselves.

To provide more flexibility in our design we modify the Origami matrix of Section 2 in two ways:

```
class ToStringV {
    public String visitLit(Lit lit) {
        return String.valueOf(lit.value));  }

    public String visitAdd(Add add) {
        return add.left.accept(this) + "+"
                + add.right.accept(this);
    }
}
interface Exp {
    String accept(ToStringV v);
}
class Lit implements Exp {
    public int value;
    Lit(int v) { value = v; }
    public String accept(ToStringV v) {
        return v.visitLit(this);  }
}
class Add implements Exp {
    public Exp left, right;
    Add(Exp l, Exp r) { left = l; right = r; }
    public String accept(ToStringV v) {
        return v.visitAdd(this);  }
}
```

Figure 5. Visitor implementation of expression problem

- We add row `Base` to put the `Exp` interface that we factor from row `Lit`. We use this row to add the visitor classes and the refinements that they make to this interface.
- We factor definition of data type classes, with their fields and constructors, into a new column `Cons`, and we place the implementation of operation `ToString` in its own column.

The new matrix with the code of the `Cons` column is shown in Figure 6.

4.2 Commonalities and Variabilities in Matrix Entries

Origami imposes a highly structured design on operations and data types, so that they are composable and one can reason about matrix designs and foldings. By imposing a structured design, we expect some uniformities in matrix entries. We can take advantage of the commonalities and variabilities that they exhibit to generate code for portions of the matrix by using frame technology [2]. A *frame* is a template that contains code fragments and a set of commands. These commands generate code particular to a variant, usually according to one or more parameters, in a process called *frame instantiation* [2][19][11]. For our two-dimensional Origami model, we define frames to capture commonalities and we define parameters to represent variations.

	Cons	ToString	Eval
Base	**bc** public interface Exp { }	bt	be

Lit	**lc**
	class Lit implements Exp {
	public int value;
	Lit (int v) { value = v ; }
	}
		lt	le

Add	**ac**
	class Add implements Exp {
	public Exp left, right ;
	Add (Exp l, Exp r) { left = l ; right = r ; }
	}
		at	ae

Neg	**nc**
	class Neg implements Exp {
	public Exp expression;
	Neg (Exp e) { expression = e ; }
	}
		nt	ne

Figure 6. New matrix with code for Cons column

4.2.1 Column analysis

In this section we study how entries vary along the columns. We observe that the code of column `Cons` does not present any significant similarities, which gives no room for frame derivation, so we focus on the operation columns `ToString` and `Eval`.

Starting with the `Base` row in Figure 6, consider the `bt` and `be` entries whose code is shown in Figure 7a and Figure 7b respectively. Both contain an empty visitor class and an `Exp` interface refinement.

A careful examination of their visitor classes reveals they are identical except for the class names, underlined in the figure. This suggests that a frame for the visitor class can be defined that is parameterized by the visitor name. Further commonality can also be identified in their refinements of the `Exp` interface. The code of each refinement is the

(a)
```
public class ToStringV { }

refines interface Exp {
 String accept (ToStringV v);
 String toString () ;
}
```

(b)
```
public class EvalV { }

refines interface Exp {
 int accept (EvalV v) ;
 int eval () ;
}
```

(c)

Columns

Variables	ToString	Eval
VISITOR	ToStringV	EvalV
RETURN	String	int
METHOD	toString	eval

(d)
```
public class ${VISITOR} {}
```

(e)
```
refines interface Exp {
 ${RETURN} accept (${VISITOR} v) ;
 ${RETURN} ${METHOD} () ;
}
```

Figure 7. Framing Base row operations

same, except they vary in the visitor class name, the return type of the operation, and name of the operation method. These variations are also underlined. This suggests a frame for `Exp` refinements that can be parameterized along these three variations.

Frame parameters are defined by variables for which we follow the convention of using all capital letters, for example **VARIABLE**, to identify a frame variable, and **${VARIABLE}** to refer to the value of a variable.

We can parameterize the variations of entries `bt` and `be` with three variables as follows:

- **VISITOR**: represents the name of the visitor class.
- **RETURN**: represents the return type of the operation.
- **METHOD**: represents the name of the operation method.

These variables represent *properties of an Origami column*, and as such are called c*olumn variables*. Their values are listed in Figure 7c. Using appropriate values for the variables, matrix entries `bt` and `be` can be constructed by instantiating the frames shown in Figure 7d and Figure 7e.

Continuing with our analysis of operation columns `ToString` and `Eval`, we now consider entries `lt`, `le`, `at`, `ae`, `nt`, and `ne` in Figure 6. Each of these entries contains two refinements: one for the visitor class, and one for the row data type. As they all have the same structure, we can pick entries of any row to study how they vary along the columns. For example, compare entries `at` and `ae` in Figure 8a and Figure 8b.We notice that they vary according to the three column variables previously identified (VISITOR, RETURN, and METHOD), whose values on both entries are underlined in the text. But they also have one more variation. This is the entry-specific code that implements the operation method for their data type, shown in *italics* in the figure. We call the code that is unique to an entry *essential code*. We use variable **OPERATION** to express this variability.

at

```
refines class ToStringV {
 public String visitAdd (Add add) {
   return add.toString () ;
 }
}

refines class Add implements Exp {
 public String accept(ToStringV v){
   return v.visitAdd (this) ;
  }
  public String toString () {
    return left.toString() + "+" +
              right.toString( ) ;
    }
}
```

(a)

ae

```
refines class EvalV {
 public int visitAdd (Add add) {
   return add.eval () ;
 }
}

refines class Add implements Exp {
 public int accept (EvalV v) {
   return v.visitAdd (this) ;
 }
 public int eval () {
   return left.eval () + right.eval () ;
 }
}
```

(b)

Figure 8. Comparing Add along columns ToString and Eval

at

```
refines class ToStringV {
 public String visitAdd (Add add) {
  return add.toString () ;
 }
}

refines class Add implements Exp {
 public String accept(ToStringV v){
  return v.visitAdd (this) ;
 }
 public String toString () {
   return left.toString() + "+" +
            right.toString( ) ;
   }
}
```

(a)

nt

```
refines class ToStringV {
 public String visitNeg (Neg neg){
  return neg.toString () ;
 }
}

refines class Neg implements Exp {
 public String accept (ToString v){
  return v.visitNeg (this) ;
 }
 public String toString() {
  return "-(" +  expression.toString() + ")" ;
 }
}
```

(b)

Figure 9. Comparing ToString along rows Add and Neg

4.2.2 Row analysis

Just as the entries of a row have commonalities that can be factored into frames and parameterized by column variables, there are commonalities in column entries that can be factored into frames and parameterized by *row variables*. Consider entry `at` in Figure 9a and compare it with entry `nt` in Figure 9b. Both contain a refinement of their visitor class that differ in the name of the data type used in the method's argument type and as suffix of the name of the visit method, and the argument of the method. The two entries also contain a refinement of their data type class that differ, not surprisingly, in their names. These variations are underlined in the figure.

These findings suggest two new frames: a refinement for the visitor class, and a refinement for the data type. These frames can be parameterized with the following variables:

- **DATA**: for the name of the data type.
- **PARAMETER**: for the name of the parameter of the *visit* method.

The information represented by these variables, changing according to the row of the matrix, is illustrated in Figure 10a.

Using appropriate values for the variables, all entries in columns `ToString` and `Eval` in Figure 6 can be constructed by instantiating the frames shown in Figure 10b and Figure 10c.

To summarize, the code encapsulated within an Origami entry can be synthesized by frames that are parameterized by column variables, row variables, and essential code variables.

4.3 Concrete Frame Implementation

To illustrate an implementation, we use a frame technology that emphasizes textual substitution; for more complex problems, a more sophisticated technology may be necessary [11][19].

(a)

Rows

Variables		Lit	Add	Neg
	DATA	Lit	Add	Neg
	PARAMETER	lit	add	neg

(b)

```
refines class ${VISITOR} {
  public ${RETURN} visit${DATA} (${DATA} ${PARAMETER}) {
    return ${PARAMETER}.${METHOD} ( ) ;
  }
}
```

(c)

```
refines class ${DATA} implements Exp {
    public ${RETURN} accept (${VISITOR} v) {
      return v.visit${DATA} (this) ;
    }
    public ${RETURN} ${METHOD}( ) {
      return ${OPERATION} ;
    }
  }
```

Figure 10. Framing operation entries

4.3.1 Frame Model

Our frame model consists of four directories described below.

Frames directory. The core of our model is a *frames directory* that contains one file for each of the four frames defined in Figure 7 and Figure 10. We follow the convention that frame files have extension `.frame` and that the base name indicates the file to be generated. For example, the file name `Exp.frame` contains the frame specification illustrated in Figure 7e and, when instantiated, that frame generates a file named `Exp.jak`. Since our model generates *only* Jak files, there is no need to specify alternative extensions.

One interesting aspect of our technique is that frame instantiation can also be applied to the file names themselves. For example, the frame shown in Figure 7d is placed in a file named `${VISITOR}.frame`. When this frame is instantiated, the generated base file name will be the value of the VISITOR variable. Similarly, the frame in Figure 10c is placed into a file named `${DATA}.frame` which, when instantiated, generates a refinement class for a data type.

However, a complicating factor arises when applying frame instantiation to generate file names. Suppose there are two or more *distinct* frames that can generate the *same* file name — how can the appropriate frame be selected? This occurs for the generated visitor classes in our model. For the `Base` row, the frame in Figure 7d must generate a base visitor class while, for the remaining rows, the frame in Figure 10b must generate a refinement of the visitor class. It is not possible for both frames to have the same file name.

Our solution is to define an additional column variable, REFVISITOR, that is assigned the same value as VISITOR. The frame in Figure 10b is then placed into a file name `${REFVISITOR}.frame`, while the frame in Figure 7d is in file `${VISITOR}.frame`.

Alternative solutions exist (e.g., supporting multiple frames directories or defining the base visitor classes manually), but ours meets the immediate needs, while simultaneously demonstrating one solution to a more general problem.

Columns directory. It includes a set of files, one for each column, containing the value assignments to all column variables. By convention, we use a file extension `.def` for definitions files and we use the column names from Figure 6 as the base filenames. Each file in the columns directory provides definitions for the column variables VISITOR, RETURN and METHOD described in Section 4.2 as well as the column parameter REFVISITOR described above. Figure 11 shows the contents of the definitions file `ToString.def`.

Rows directory. Includes a set of definitions files, one for each row. The base file names are taken from the row names in Figure 6 and each file provides definitions for the row variables DATA and PARAMETER. Figure 11 shows the contents of the definitions file `Add.def`.

Entries directory. Contains subdirectories, one for each matrix entry. Their names follow the naming convention *row_column*. These subdirectories contain constant files, copied verbatim to the target matrix entry, and a definitions file with the same name of the subdirectory. Each entry definitions file specifies the essential code definitions, for our example, in the variable OPERATION. It also defines, via a `generates` statement, the frames that must be instantiated in that entry. Figure 11 shows the contents of the `Add_ToString.def` file.

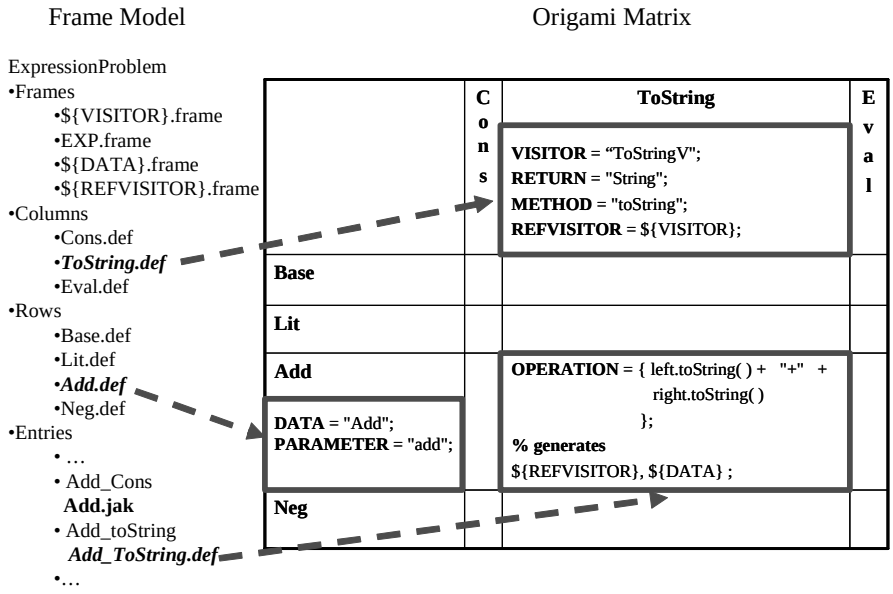

Figure 11. Relationship between Frame Model and Origami Matrix

4.3.2 Frame instantiation.

The purpose of our frame model is to generate the code for the Origami matrix shown in Figure 6 and described in Section 4. Instantiation starts by determining the set of column names and the set of row names. This is done by examining the base names of the definitions files in the columns directory and the rows directory, respectively. Then, for every possible pairing of row name with column name, the following steps are taken:

- Create a symbol table with the parameter definitions taken from the corresponding definitions files in the rows directory, the columns directory and the entries directory. In our model, there is no possibility of conflicting parameter names (e.g., a row variable and column variable with the same name). However, in general, tools should provide conflict resolution, either by reporting conflicts as errors or by overriding broader definitions with more specific ones.
- In each entry subdirectory: For each frame file specified in the `generates` statement of the definitions file, instantiate that frame by textually substituting the parameter definitions into both the frame file's name and the frame file's contents. The result is then written into an entry-specific subdirectory of a target directory specified via a command-line argument. The constant files of the entry's subdirectory are copied verbatim to the same subdirectory.

Once the code is generated into the target directory, it represents a standard Origami model that can be composed as described in Section 3.

5 Evaluation

We implemented the example as described in the previous section to validate our ideas, and also to understand better where our technique could be most practical. As the expression problem is small, we do not have a code base from which to extrapolate realistic quantitative benefits for our technique.

However, from our experience with large Origami-based generators, where synthesized programs exceed the equivalent of 30K Java LOC, the key issue is *code base consistency*. As we have seen with the expression problem, the code for matrix entries are parametrically related. As the number of dimensions and the number of units per dimension grow, automated maintenance of these consistency relationships becomes important.

For example, for us to manually create a new column or row in a matrix requires manual modification that is both tedious and error-prone. Our research in program generation suggests that we consider tool support to eliminate these common and rote activities. The commonality and variability analysis that we have described is clearly part of any tool support for solving the consistency maintenance problem.

Further, to make our technique practical — meaning that it can scale to much larger code bases — requires frames to be produced *automatically*; the overhead in *manually* creating frames and the infrastructure to provide code synthesis and consistency maintenance is otherwise too large to realize an overall benefit.

Tool support could be provided to extend a dimension under the guidance of a developer. For example, the addition of a new column in a two-dimensional model could be supported by extracting a pre-existing column as a *prototype*. Our hypothetical tool could copy the prototype to the new column, while parametrically updating it and removing entry-specific code. Similar support can be provided for the other dimensions of a model. Thus, a consistent code template for a set of co-designed matrix entries can be generated. Programmers would then complete these templates by specifying the values of essential code.

Issues of identifying row and column (or more generally, dimensional) variables could also be simplified by tool support. This could be done by allowing programmers to highlight code fragments and to bind their values to particular dimensional variables. Thus, when code is copied, a supporting tool could infer the new matrix coordinates from the specific variables and substitute the appropriate parametric values.

In effect, we envision a class of tools comparable to language editors, which complete code fragments and programmatic phrases as programmers enter their code. Such editors would understand the structure of the permitted code models and would automatically complete parts of the program by inference, thereby relieving programmers of unnecessary burdens. The tools that we envision would enforce necessary consistencies across large designs. The underlying technology by which this would be achieved is the frame technology that we have outlined in this paper.

6 Related Work

Related work is categorized as concrete or abstract. The concrete category contains the two technologies that we have used: *Frames technology* [2][11][19] and the *AHEAD product line* [4][5][6]. These technologies address, respectively, code generation and composition, which we have combined to implement our multi-dimensional frame model. A body of literature, some of which we have cited, exists to explain these technologies in depth. We refer the reader to this literature.

However, the abstract category of related work contains the primary concept that motivated our work: *Multi-dimensional separation of concerns*. In general, separation of concerns [13][14] is a motivating idea behind modularization and encapsulation in software designs. Concerns can range from the purely technical (e.g., the support of separate compilation and module replacement) through developmental (e.g., the support of comprehensibility and independent development) to include system and behavioral concerns (e.g., coupling and communication in distributed environments).

Early programming languages and disciplines addressed the purely technical concerns of separate compilation and module re-assembly, but later software development trends attended to increasingly higher-level concerns. Parnas, in two key papers [13][14], specified the use of *information hiding* as a modularization criterion and he also discussed the dependence of modularization on the social or managerial support of a software product. Object-oriented programming methodologies address concerns such as these by directly supporting the modularization of functions with related data and by encouraging the encapsulation of object collaborations as patterns [10].

Tarr *et alia*[16] extends these trends by advocating the *multi-dimensional* separation of concerns. The goal is to simultaneously separate overlapping concerns along multiple dimensions — conceptually, dimensional separation can be viewed as a type of encapsulation. Once achieved, each dimension of concern can, ideally, be understood and extended with little impact on other dimensions. To achieve this goal, mechanisms must be developed that support decomposition along these dimensions along with the subsequent composition into programs and product-line members.

We have described one such mechanism. Decomposition is explicated by the development of a multi-dimensional frame model and composition is implemented by a tool chain, comprising both frame instantiation and AHEAD composition, that generates a product from a multi-dimensional frame model.

7 Conclusions

Multi-dimensional feature sets provide a compact and highly-structured way to express product-line designs as matrices. Structured designs introduce uniformities (and hence consistency problems) into the code base, where the source code of matrix entries is highly parametrically related. Eliminating the tedium of manually framing code commonalities and, more importantly, of maintaining the consistency of matrix entries is both challenging and interesting.

In this paper, we explained how multi-dimensional feature sets have an elegant representation as Origami matrices. We used the expression problem to illustrate the key issues of commonality extraction and consistency maintenance. We showed how the parametric relationships among matrix entries can be captured by frame technologies, and how code synthesis and consistency maintenance could be realized.

The contribution of this paper is to take the first step towards solving this problem in the large. We demonstrated that its solution requires commonality and variability analysis, and that code synthesis can be realized by frame technologies. This, however, is only the first step. Our work has shown that to large-scale application of these techniques will require a class of program development tools to automate the development of frames, to synthesize code templates for matrix entries, and to maintain consistency constraints across matrix models. We believe the potential for such tools is significant, for no other reason that a core problem in program generation is elegantly captured by Origami matrices — that applications will always need to be extended by new data types and operations, and tool support for creating and maintaining such designs is essential.

Acknowledgements. We thank Don Batory for his comments and recommendations on drafts of this paper.

8 References

[1] AHEAD Tool Suite (ATS). http://www.cs.utexas.edu/users/schwartz

[2] Bassett, P.G.: Framing Software Reuse. Lessons from the Real World. Yourdon Press Computing Series, Prentice Hall (1997)

[3] Batory, D., Geraci, B.J.:Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering, February (1997) 67-82

[4] Batory, D., Lopez-Herrejon, R., Martin, J.P.: Generating Product-Lines of Product-Families. Automated Software Engineering Conference, September (2002)

[5] Batory, D., Liu J., Sarvela, J.N.:Refinements and Multidimensional Separation of Concerns. Foundations on Software Engineering FSE, September (2003)

[6] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. International Conference on Software Engineering ICSE, May (2003)

[7] Cook, W.R.: Object-Oriented Programming versus Abstract Data Types. Workshop on Foundations of Object-Oriented Languages, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178

[8] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

[9] Findler, R.B, Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. International Conference on Functional Programming ICFP, (1998) 94-104

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse of Object-Oriented Designs. Addison-Wesley (1994)

[11] Holmes, C., Evans, A.: A Review of Frame Technology, Technical Report YCS-2003-369, York University, UK (2003)

[12] Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing object-oriented and functional design to promote re-use. European Conference on Object-Oriented Programming ECOOP. Lecture Notes in Computer Science, Vol. 1445, (1998) 91-113

[13] Parnas, D. L.: Information Distribution Aspects of Design Methodology. Technical Report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, (1971)

[14] Parnas, D. L.: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), December (1972) 1053-1058

[15] Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. European Conference on Object-Oriented Programming ECOOP. Lecture Notes in Computer Science, Vol. 1241, (1997) 419-443

[16] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. International Conference on Software Engineering ICSE, (1999) 107-119

[17] Torgensen, M.:The Expresion Problem Revisited. Four new solutions using generics. To appear European Conference on Object-Oriented Programming ECOOP (2004)

[18] Wadler, P. The expression problem. Posted on the Java Genericity mailing list (1998)

[19] Zhang, H., Jarzabek, S., Swe, S. M.: XVCL Approach to Separating Concerns in Product Family Assets. International Conference on Generative and Component-Based Software Engineering. Lecture Notes in Computer Science, Vol. 2186, (2001) 36-47