Accepted for publication in International Conference on Software Engineering, May 2003, Portland, Oregon

Scaling Step-Wise Refinement

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

Axel Rauschmayer
Ludwig-Maximilians-Universität München
Institut für Informatik
D-80538 Munich, Germany
axel@rauschma.de

Abstract[1]

Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by adding features incrementally. We present the *AHEAD (Algebraic Hierarchical Equations for Application Design)* model that shows how step-wise refinement scales to synthesize *multiple programs* and *multiple non-code representations*. AHEAD shows that software can have an elegant, hierarchical mathematical structure that is expressible as nested sets of equations. We review a tool set that supports AHEAD. As a demonstration of its viability, we have bootstrapped AHEAD tools solely from equational specifications, generating Java and non-Java artifacts automatically, a task that was accomplished only by ad hoc means previously.

1 Introduction

Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by incrementally adding details [9]. The program increments that we consider in this paper are *feature refinements* — modules that encapsulate individual features where a feature is a product characteristic that is used in distinguishing programs within a family of related programs (e.g., a product-line) [13].

There are many implementations of feature refinements, each with different names, capabilities, and limitations: layers [2], feature modules [17], meta-classes [10], collaborations [23][24], subjects [14], aspects [18], and concerns [27]. More general than traditional modules that encapsulate sets of complete classes, a feature refinement usually encapsulates *fragments* of multiple classes. Figure 1 depicts three classes, `c1`—`c3`. Refinement `r1` *cross-cuts* these classes, i.e., it encapsulates fragments of `c1`—`c3`. The same holds for refinements `r2` and `r3`. Composing refinements `r1`—`r3` yields a set of fully-formed classes `c1`—`c3`. Because refinements reify levels of abstraction, feature refinements are often called *layers* — a name that is visually reinforced by their vertical stratification of `c1`—`c3` in Figure 1. As the concepts of refinements, layers, and features are so closely related, we use their terms interchangeably. In general, feature refinements are modular, albeit unconventional, building blocks of programs.

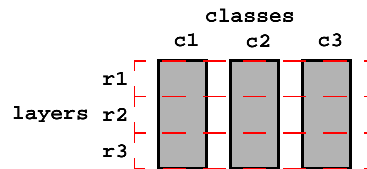


Figure 1. Classes and Refinements (Layers)

Tools that synthesize applications by composing feature refinements are *generators* whose focus has been on the production of source code for individual programs. This is too limited. Today's systems are not individual programs, but rather groups of different programs collaborating in sophisticated ways. Client-server architectures are examples, and so are tool suites, such as MS-Office. Further, systems are not solely described by source code. Architects routinely use different knowledge representations (e.g., process models, UML models, makefiles, design documents) to capture an application's design [16]. Each representation encodes different design information and is expressed in its own language or *domain-specific language (DSL)*, such as UML, OCL, XML, etc.

The contribution of this paper shows how step-wise refinement scales to the simultaneous synthesis of multiple programs and multiple non-code representations written in different DSLs. The challenge is not one of possibility, as ad hoc ways are used now. Rather, the challenge is to create a mathematical model of application synthesis that treats all representations — code and non-code, individual programs and multiple programs — in a uniform way. By expressing the refinement of representations as equations, we not only simplify tool development (as equations are ideal for program manipulation) but also lay the groundwork for specifying, generating, and optimizing application designs of considerable complexity using algebraic techniques.

1. This work was supported in part by the U.S. *Army Simulation and Training Command (STRICOM)* contract N61339-99-D-10 and *Deutsche Forschungsgemeinschaft (DFG)* project WI 841/6-1 "InOpSys".

We begin with a review of the GenVoca model, which shows how the *code* representation of an *individual* program is expressed by an equation. We then present the *Algebraic Hierarchical Equations for Application Design (AHEAD)* model that generalizes equational specifications to multiple programs and multiple representations. AHEAD is related to other methodologies, such as Aspect-Oriented Programming [18] and Multi-Dimensional Separation of Concerns [22], and thus our results are not GenVoca-specific. We review a tool set that supports AHEAD. As a demonstration of AHEAD's viability, we have bootstrapped AHEAD tools, generating over 100K LOC of Java (and other non-code artifacts) solely from equational specifications automatically, a task that was accomplished only by ad hoc means previously. We believe that this is the most sophisticated system built by automated step-wise refinement.

2 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 [2]. It follows traditional step-wise refinement [9] 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.

2.1 Model Concepts

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 GenVoca *model* of a domain a set of constants and functions. 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 casually 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 *expression optimization* to determine which implementation of `k` is best (e.g., provides the best performance)[2]. It is possible to automatically design software (i.e., produce an expression that optimizes some criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning — historically called *automatic programming* [1] — is [5].

Although GenVoca constants and functions seem untyped, typing constraints do exist. *Design rules* 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. The details of design rules are not germane to our paper and can be found in [3].

2.2 Model Implementation

A GenVoca constant is a set of classes. Figure 2 depicts a constant `i` that encapsulates four classes (a_i—d_i). A GenVoca function is a set of classes and class refinements. A *class refinement* adds new data members, methods, and extends or overrides existing methods of a target class. It could be implemented as a subclass of the target class, *with the provision that the subclass assumes the name of the target class*. In this sense, class refinement is different from typical subclassing in that true subclasses have names that are distinct from their parent class.

Figure 2 shows the result of applying function `j` to `i`: three classes are refined and another class is added, where the vertical lines in Figure 2 denote class refinement. That is, `j` encapsulates a cross-cut that refines classes a_j, c_j, and d_j, and adds class e_j. Figure 2 also shows the application of function `k` to `j(i)`, resulting in the refinement of two classes. In general, a forest of inheritance hierarchies is created as features are composed, and this forest grows progressively broader and deeper as the number of features increase [6].

Linear inheritance chains, called *refinement chains*, are common in this method of implementation. The rule is that

2. Different expressions represent different programs and expression 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.

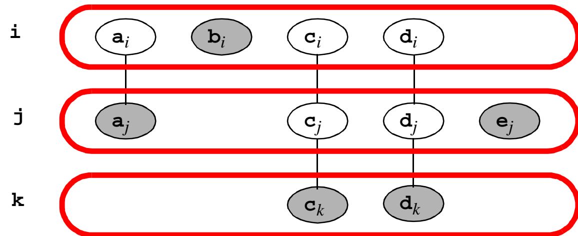


Figure 2. Simultaneous Refinement of Classes

only the bottom-most class of a chain is instantiated, because this class implements all capabilities that were assigned to it. These classes are shaded in Figure 2. For example, the bottom-most class of the `c` refinement chain implements the capabilities defined in c_i, c_j, and c_k from features `i`, `j`, and `k` respectively. Stated another way, c_i is a class (constant) and c_j and c_k are class refinements (functions). The class `c` that is produced in Figure 2 can be expressed equationally as $c = c_k(c_j(c_i))$.

The simplest implementations of feature refinements are templates [24][29]. More sophisticated ways include generators [5][20], program transformations [8], and objects [26].

3 Scaling Refinements

Four ideas have led us to a major generalization of the GenVoca model. First, a program has many different representations besides source code. A program can be defined by UML documents, process diagrams, makefiles, performance models, design rule files, etc. in addition to code, each written in its own DSL.

Second, conventional notions of modularity must be broadened: *a module is a containment hierarchy of artifacts*. For example, an object-oriented class is a module; it encapsulates a set of data members and methods. A package is another kind of module; it encapsulates a set of `Java` and `.class` files. A J2EE EAR file is also a module; it encapsulates Java JAR files, deployment descriptors, and web archive files (e.g., HTML files). A program is yet another kind of module; its representation is expressible as a containment hierarchy of artifacts — code, makefiles, etc.

Third, the impact of refinements need not be limited to source code. When a program is refined, any or all of its representations may be changed. This is required so that all of its artifacts (code files, design rules, etc.) remain consistent. Thus if a program is represented by a containment hierarchy, a program refinement is a function that transforms containment hierarchies. That is, a refinement may alter a containment hierarchy by adding new nodes (e.g., a refined program may have new `Java` or HTML files) and refining existing nodes (e.g., existing `Java` and HTML files are updated).

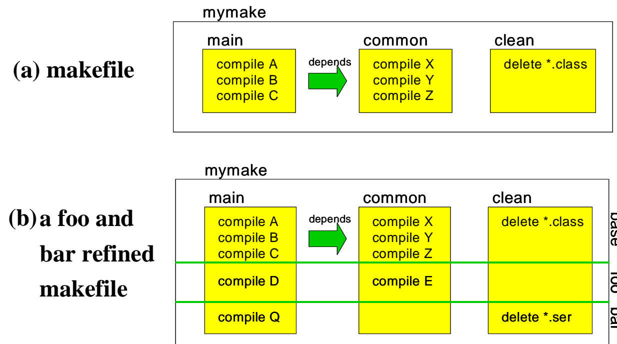


Figure 3. Makefiles and their Refinements

Fourth, refining non-code artifacts is intuitively evident: you start with an original document and you produce an updated document. But how is this to be achieved? What general principle guides refinement? An example reveals a basic strategy that we have found to be practical. A makefile is a typical non-code artifact. Figure 3a shows a makefile with three targets: `main`, `common`, and `clean`. `common` must be built before `main`; `clean` has no dependencies. Suppose these targets are part of a `base` feature. Figure 3b shows the refinement of `base` by `foo`, which encapsulates a cross-cut of targets that adds a file `D` to compile in `main` and file `E` to compile in `common`. Figure 3b shows a further refinement of `foo(base)` by `bar`, which adds another file `Q` to compile to `main` and a new instruction (`delete *.ser`) to `clean`.

Figure 3 imposes a class-like structure on a makefile. That is, a makefile is analogous to a class and targets are analogous to methods. Refinement of makefiles refines targets and may add new targets, etc. This example illustrates the *Principle of Uniformity*: treat all non-code artifacts as classes, and refine them analogously.

Interestingly, most artifact types in use today (XML, HTML, PPT, etc.) have or can have a class-like structure and thus are *object-based*. However, these types are rarely *object-oriented* — few allow inheritance or more general refinement relationships to be defined among its files. There is, for example, no support for inheritance relationships to relate different XML schemas. For representations like XML, adding inheritance or refinement relationships seems reasonable as a plethora of tools for manipulating XML documents are available and can be used to define such relationships. However, other representations — MS Word for example — are so complex that defining inheritance relationships among Word documents seems daunting. Never the less, the idea is clear: the task to be done when introducing a new artifact type is to define what refinement

means.[3] Given the ability to refine non-code artifacts, we are in the position to scale refinements in a significant way.

4 AHEAD

GenVoca expressed the *code* representation of an *individual* program as an equation. The model that we present here, called *Algebraic Hierarchical Equations for Application Design (AHEAD)*, expresses an arbitrary number of programs and representations as nested sets of equations, a form that is ideal for generators to manipulate. In this section, we show how AHEAD constants, functions, and their compositions are represented, and illustrate the power of the model. In Section 5, we review a tool set for AHEAD.

4.1 Constants and Functions

Base artifacts are *constants* and artifact refinements are *functions*. An artifact that results from a refinement chain is modeled equationally as a series of functions (refinements) applied to a constant (base artifact).

Figure 4a depicts our graphical notation for a GenVoca constant `f` that encapsulates base artifacts (henceforth called *units*) a_f—c_f. Instead of pictures, we express constant `f` mathematically as a set of constants: $f = \{ a_f, b_f, c_f \}$. Similarly, Figure 4b depicts our graphical notation for a GenVoca function `h` that encapsulates functions a_h and b_h and constant d_h. Function `h` can be expressed mathematically as a set of functions and constants: $h = \{ a_h, b_h, d_h \}$.

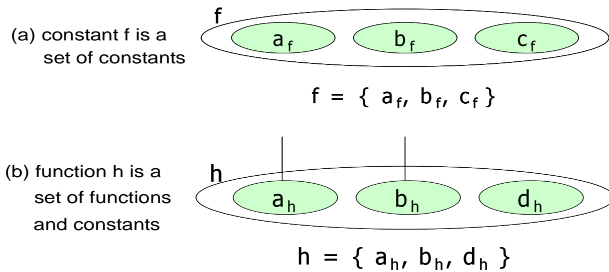


Figure 4. Constants and Functions as Sets (Collectives)

We use the term *collective* as an alternative to “set of units”. The root collective (that comprises all other collectives) is also called a *model*.

4.2 Composition

Instead of writing `h(f)` to denote the composition of `h` with `f`, we henceforth write $h \bullet f$. The composition of collectives is governed by the rules of inheritance. Namely, all units of the parent (inner or right-hand-side) collective are inherited by the child (outer or left-hand-side) collective. Further, units with the same name (ignoring subscripts) are composed pairwise with the parent unit as the innermost term:

$$
\begin{aligned}
h \bullet f &= \{ a_h, b_h, c_h \} \bullet \{ a_f, b_f, d_f \} \\
&= \{ a_h \bullet a_f,\ b_h \bullet b_f,\ c_h,\ d_f \}
\end{aligned}
$$

Equivalently, $h \bullet f$ is a set of equations where equation names are unit names without subscripts:

$$
\begin{aligned}
h \bullet f &= \{ a, b, c, d \} \\
\text{where } a &= a_h \bullet a_f \\
b &= b_h \bullet b_f \\
c &= c_h \\
d &= d_f
\end{aligned}
$$

Each expression in a collective defines the refinement chain of an artifact that is to be produced. Figure 5 shows this correspondence between our graphical notation and its AHEAD expression: $a_h \bullet a_f$ is the refinement chain for artifact `a` and $b_h \bullet b_f$ is the chain for `b`. Artifacts `c` and `d` are unrefined from their original definitions.

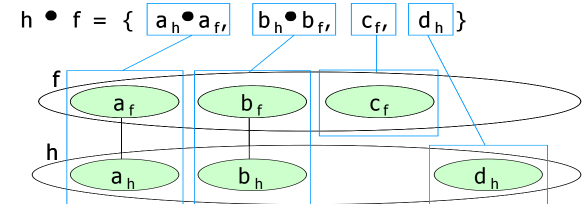


Figure 5. Expression and Refinement Chain Correspondence

Containment Hierarchies. Compound artifacts are expressed by recursion: units may themselves be collectives. Composition of compound artifacts is achieved by recursively composing collectives. Suppose a_h and a_f in Figure 5 are the collectives $a_h = \{ x_h, z_h \}$ and $a_f = \{ x_f, y_f \}$. The expression $a_h \bullet a_f$ expands to the collective $\{ x_h \bullet x_f, y_f, z_h \}$. The depth to which collectives are nested is the *rank* of the collective. `{}` is an empty collective of rank 0; `{{}}` is a collective of rank 1, and so on.

A compound artifact is a containment hierarchy. A Java program `p`, for example, is a compound artifact and thus is a non-leaf node; its leaves might be the set of Java files {`x.java`, `y.java`, …} that implement it, the set of HTML files {`x.html`, `y.html`, …} that `JavaDoc` produces from these Java files, the set of `.class` files {`x.class`, `y.class`, …} produced by `javac`, and so on.

A feature — whether it represents an AHEAD constant or a function — is defined by a tree of artifacts. When features are composed, all of their corresponding artifacts are com-

3. Because it is possible to derive artifacts (e.g., `.class` files are derived from `Java` files via `javac`), composition operators are needed only for the most basic artifact types. Maintaining the consistency among “basic” types (e.g., English explanations of source code) is something that is accomplished manually.

posed. Thus, feature composition has a simple interpretation.

Polymorphism. The composition operator ● is polymorphic. Artifacts are composed by operators that are specific to that artifact type. Java files, design rule files, XML files, etc. will each have their own unique implementation of the ● operator. Thus, if there are *n* different artifact types, there will be *n* different type-specific composition operators. (As we will see in Section 5, there may be several composition operators for a given artifact type).

Further, the operator for composing collectives of rank *n* is no different than the operator for composing collectives of rank $n+1$, for $n \geq 0$. We call this the *Principle of Abstraction Uniformity*, which is a special case of the Principle of Uniformity. Imposing uniformity on all levels of abstraction produces a very compact and powerful algebra for defining and composing systems.

Scalability. OO programming languages that support parameterized inheritance can define and refine code artifacts, but are unsuitable for non-code representations. In contrast, equations elegantly express refinement relationships for *all* representations. Furthermore, *equations enable step-wise refinement and its generators to scale*. Instead of building one huge generator that deals with all possible program representations (which itself is impractical), it is *much* easier to build an elementary tool — here called a `composer` — that expands a high-level equation into its constituent artifact equations. The `composer` can then submit its generated equations to type-specific composition tools to generate the artifacts of the target system. Thus, a simple `composer` tool does the work of orchestrating other relatively simple tools to produce the complex set of artifacts that comprise a synthesized system.

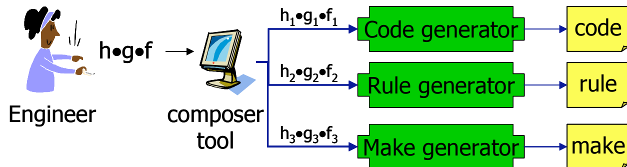


Figure 6. Organization of AHEAD Generators

4.3 Metamodels

One way in which to gauge the power of a model is its ability to express itself. A *metamodel* is a model whose instances are models. The UML metamodel is a classical example: it is a UML model whose instances are other UML models. AHEAD metamodels are collectives of models. Consider model `M` that contains three units `a`—`c`:

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

Metamodel `MM` contains three units `AAA`—`CCC`, each of which is a collective with a single unit:

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

Model `M` is defined by an equation using metamodel `MM`:

```
M = AAA ● BBB ● CCC
```

where composition creates `M` by disjoint union, since none of the units share names. The following are interesting uses of metamodels.

Service Packs. A *service pack* is an update of model. A *service pack metamodel* `SP` contains an initial model M_0 and a series of service pack updates S_1, S_2, S_3, each of which incrementally updates a model:

$$SP = \{ M_0, S_1, S_2, S_3, \ldots \}$$

A special composition operator ♦, called *replace*, is used to apply the changes of a service pack to an existing model. If U_k and U_j are primitive (i.e., non-compound) units, the law of the replace operator is:

$$U_k \blacklozenge U_j = U_k \quad \text{// } U_k \text{ replaces } U_j$$

Otherwise ♦ is identical to the composition ● operator for collectives. (♦ is actually a special case of ●). Thus, a model `M` that is up-to-date w.r.t. service pack S_3 is defined by the equation:

$$M = S_3 \blacklozenge S_2 \blacklozenge S_1 \blacklozenge M_0$$

That is, the effects of S_1 are applied to M_0 by replacing old base artifacts with new ones, and adding new artifacts. The same for S_2 and S_3. Special primitive units might be used to indicate the physical deletion (rather than replacement) of designated files.

Origami. A much more sophisticated use for AHEAD metamodels is generating equations for AHEAD tools from a single AHEAD metamodel equation. See [7].

5 AHEAD Tool Support

A simple way to implement a collective is as a file system directory. A directory's contents are either files (primitive units) or subdirectories (collectives). Figure 7a shows a collective that defines layer `A`; it consists of a unit, `R.drc` (a design rule file), and two collectives, `Code = {X.jak, Y.jak}` (Jak files are extended-Java files) and `Htm = {W.htm}`. Figure 7b depicts its representation as a directory.

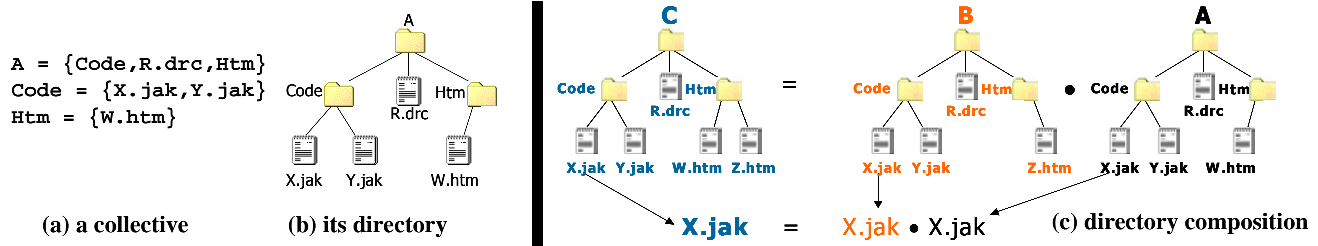


Figure 7. Collectives as Directories

Feature composition is directory composition. A composite directory is produced whose organization is isomorphic to the directories from which it was composed. Figure 7c shows a composition of features `A` and `B`, the result is feature `C`. Corresponding units in each directory are composed to produce a compound unit. For example, `X.jak` of `C` equals `X.jak` of `B` composed with `X.jak` of `A`.

Given this organization, we have built many tools to support AHEAD using the *Jakarta Tool Suite (JTS)* [4]. JTS is a set of compiler-compiler tools that use GenVoca models to build *product-lines* (i.e., families) of preprocessors for extended Java languages. Almost all AHEAD tools were built using JTS, and were written in a JTS-produced dialect of Java that includes refinement declarations, meta-programming constructs (e.g., Lisp "quote" and "unquote"), and hygienic macros. In Section 6 we describe how we recently bootstrapped our implementation where JTS is no longer used and AHEAD tools are now creating AHEAD tools. AHEAD tools are, themselves, Java language extensible — they can be created for different dialects of Java through language feature compositions.

The main tool of AHEAD is the `composer`, which takes an equation as a command-line input, and recursively expands the equation into its nested collective equivalent. It then creates a composite feature directory (whose name is that of the input equation), and invokes artifact-specific composition tools to synthesize artifact files from generated nested equations. `composer` itself is fairly simple, written in 4K lines of Java.

Other tools implement the composition operator • for specific artifact types. The first tools that we built composed code artifacts, as verifying the code synthesis capabilities of AHEAD was our first priority. Subsequently, composition tools for HTML files, makefiles, equation files (files whose contents define a single AHEAD equation), design rule files, and BNF-grammar files (for synthesizing extensible preprocessors) were constructed. We anticipate the number of artifact-composition tools to increase over time.

Because of space limitations, we review how AHEAD tools compose code and equation representations. Other tools and representations — such as makefiles and design rules — are described in [28].

5.1 Code Artifacts and Tools

Code files that are composed by AHEAD tools are not pure Java, but rather a superset of Java called Jak (pronounced "jack", short for Jakarta): this is Java extended with embedded DSLs for refinements, state machines, and meta-programming. (Remember: AHEAD tools are Java language extensible, so AHEAD can support many Java dialects).

Jak-specific tools are invoked to compose Jak files. One of two different implementations of the • operator can be used: `jampack` or `mixin`. Both take an equation as input, which defines the refinement chain of a Jak artifact, and produce a single, composite Jak file as output. A third tool, `jak2java`, translates a Jak file to its Java counterpart. Thus our two-step paradigm uses `jampack` or `mixin` to compose Jak files, and `jak2java` to derive the corresponding Java file from its composite Jak file (Figure 8).

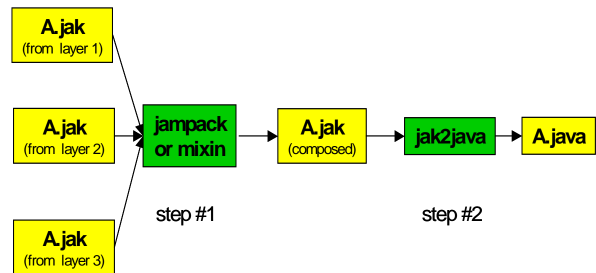


Figure 8. Composing and Translating Jak files

Source Code. A Jak file defines a code constant or function. A code constant is a single interface, class, or state machine. A Jak interface and class declaration are indistinguishable from their Java counterparts, except for a layer declaration which specifies the name of the layer to which the file belongs (see Figure 9a). More interesting is a state machine declaration, which consists of state and edge (transition) declarations. The state machine `fsm` of Figure 9b declares three states (`s1`—`s3`) and two edges (`e1`—`e2`). See [6] for more details.

```
layer A;

import java.util.*;

class k {
   int counter;

   int getCounter()
   {...}
}
```

(a) class constant

```
layer A;

import java.util;

State_machine fsm {
   States s1, s2, s3;

   Edge e1: s1 -> s2 ...;
   Edge e2: s2 -> s3 ...;
}
```

(b) state machine constant

```
layer B;

import foo.bar;

refines class k {
   int counter2;

   void method2()
   {...}
}
```

(c) class function

```
layer B;

import foo.bar;

refines State_machine fsm {
   States s4;

   Edge e3: s3 -> s4 ...;
}
```

(d) state machine function

Figure 9. AHEAD Code Constants and Functions

A code function refines an interface, class, or state machine. The `refines` modifier distinguishes constant declarations from functions. A refinement of class `k` in Figure 9a is shown in Figure 9c. It adds a new data member (`counter2`) and a new method (`method2`). Methods can be refined/extended in the usual way by overriding their superclass method and invoking that method in the extension body. A refinement of the `fsm` state machine is shown in Figure 9d. It adds another state `s4` and edge `e3` declaration.

Composition. `jampack` and `mixin` are sophisticated tools, both over 30K Java LOC in size. Their basic functionality, however, is fairly simple. `jampack` compresses refinement chains into a single interface, class, or state machine specification. The `jampack` result of composing the `k` function with the `k` constant of Figure 9 is shown in Figure 10a; the resulting class exposes the union of package imports, data members, and methods that are visible to the bottom-most class of its refinement chain. The `jampack` result of composing the `fsm` function with the `fsm` constant of Figure 9 is shown in Figure 10b. The resulting state machine exposes the union of package imports, states and edges, that are visible to the bottom-most state machine of its refinement chain.

`jampack` was our first tool to compose code. We soon discovered that `jampack` might not be the preferred tool. To see why, a typical debugging cycle is to (a) compose Jak files, (b) translate the composite Jak file to its Java counterpart, (c) compile and debug the Java file, and (d) update the composite Jak file with bug fixes. This translate-compile-update cycle continues until no further changes are needed. Here lies the problem: `jampack` does not preserve layer boundaries, thus changes made to the composite Jak file must be manually propagated back to the original Jak layer files. This can be tedious and error prone.

```
layer C;

import java.util.*;
import foo.bar

class k {
   int counter;
   int counter2;

   int getCounter()
   {...}
   int method2()
   {...}
}
```

(a)

```
layer C;

import java.util.*;
import foo.bar;

State_machine fsm {
   States s1, s2, s3;
   States s4;

   Edge e1: s1 -> s2 ...;
   Edge e2: s2 -> s3 ...;
   Edge e3: s3 -> s4 ...;
}
```

(b)

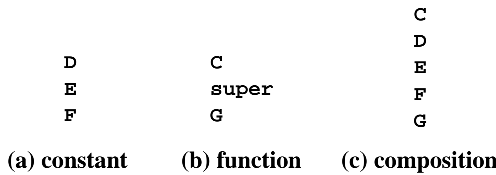
Figure 10. `jampack` Compositions of Jak Files

`mixin` was created as an alternative implementation of the Jak composition operator. It preserves layer boundaries by creating within a single Jak file an inheritance hierarchy — refinement chain — of class/interface/state machine declarations, where the bottom-most declaration is concrete and all others are `abstract`. Every declaration is prefaced by a `SoUrCe` statement that identifies both the layer name and path to the file from which that instantiated declaration originated. (The case alteration in "`SoUrCe`" minimizes the likelihood of conflicting with typical program identifiers). Figure 11a-b shows the `mixin` output for the corresponding `jampack` compositions of Figure 10a-b. With the availability of `SoUrCe` statements, we have created a fourth tool, called `unmixin`, which takes a `mixin`-produced Jak file as input, and automatically propagates updates (comments as well as source statements) to the original layer files. Early experiences with `unmixin` suggest it can save a considerable amount of time during a development cycle.

There is yet another strategy: it is possible to edit the original layer files and recompose. Our experience suggests that debugging layers is similar to debugging templates: one instantiates a template to debug and develop it, and changes are eventually back-propagated to the original definition. We do not yet know which tools and processes engineers will prefer when developing software with AHEAD; this is a subject of future work.

5.2 Equation Artifacts

An *equation file* encodes an AHEAD equation. The file for `X = F•E•D` is depicted in Figure 12a; the name of the file is `X.equation` and the file itself is a text file that lists one layer per line, inner-most layer first. A refinement of an equation follows the principle of uniformity: we treat an equation as an artifact and a refinement is an equation that may reference a special layer called `super`, which refers to the parent equation. A refinement of an equation is depicted in Figure 12b, and the result of composing the equation files of Figure 12a-b is Figure 12c. If a refinement does not reference `super`, it is a constant and overrides the parent definition. Depending on the replacement of `super` in an equation file, a refinement might add new layers before, after, or around the original equation file (like before, after, and around methods).



(a) constant (b) function (c) composition

Figure 12. Equation Files

Equation files are useful as command-line input to `composer` and for implementing metamodels [7].

6 Applications

AHEAD is being used to build next-generation distributed *fire support simulators (FSATS)* for the U.S. Army *Simulation, Training, and Instrumentation Command (STRICOM)*. Several years ago, we built a layered prototype of FSATS [6]. As the first non-trivial test of AHEAD, we converted the prototype's source code, which included classes, interfaces, and state machines, into AHEAD layers. We also added design rule files and makefiles, so when AHEAD layers are composed, three very different representations of FSATS (code, design rules, makefiles) were synthesized. The prototype was defined by a single AHEAD equation composing 21 layers, yielding 90 files and 4500 Java LOC.

```
layer C;

import java.util.*;
import foo.bar;

SoUrCe A "A/k.jak";
abstract class k001 {
   int counter;
   int getCounter()
   {...}
}

SoUrCe B "B/k.jak";
public class k
   extends k001 {
   int counter2;
   void method2()
   {...}
}
```

(a)

```
layer C;

import java.util.*;
import foo.bar;

SoUrCe A "A/fsm.jak";
abstract State_machine
   fsm001 {
   States s1, s2, s3;
   Edge e1: s1 -> s2 ...;
   Edge e2: s2 -> s3 ...;
}

SoUrCe B "B/fsm.jak";
public State_machine fsm
   extends fsm001 {
   States s4;
   Edge e3: s3 -> s4 ...;
}
```

(b)

Figure 11. `Mixin` Compositions of Jak Files

The next significant test was bootstrapping AHEAD itself. As mentioned earlier, AHEAD tools were initially built using JTS. To bootstrap AHEAD, we converted JTS source into AHEAD layers. In addition to code representations, AHEAD directories included grammar (BNF) files, which defined the syntax of optional extensions to the Java language. We used Origami, an AHEAD metamodel, to generate equations for AHEAD tools, including `jampack`, `mixin`, `unmixin`, and `jak2java`, and then used AHEAD tools to synthesize their executables [7].

To convey the complexity of this bootstrapping step, there are 69 distinct AHEAD layers constituting a code base of 33K Jak LOC. An Origami-generated equation references approximately 23 layers, where about 10 layers are shared among AHEAD tools. Each tool is generated by composing the Jak and grammar representations of each layer, and translating their representations to Java. Each AHEAD tool is sizable, having over 30K Java LOC. Thus using only equational specifications, we are generating the AHEAD tool suite whose complexity exceeds 100K Java LOC [7] *automatically*, a task that was accomplished only by ad hoc means previously. We believe this is the most sophisticated system ever produced by automated step-wise refinement.

7 Future Work

AHEAD raises many interesting questions including:

- The ability to analyze designs using tools such as model checkers will be critical to future design technologies. How do such tools fit with refinements and AHEAD designs? Preliminary results are encouraging [19].
- There are *many* operators on collectives and units, besides the • and ♦ operators. `javac`, for example, is an operator that compiles the Java files in a collective. `javadoc` is an operator that generates HTML documentation for the Java files in a collective. By equating standard tools with operators on collectives, we have an algebra for software development. Once software is specified algebraically, it is amenable to automated optimization and reasoning.
- How do the operators of refactorings and program optimizations fit into an AHEAD algebra?
- Many refinements that impact — cross-cut — different parts of programs require more sophisticated implementations of refinements than used in AHEAD. For exam-

ple, information from multiple artifacts may be used to decide how to refine other artifacts en mass. How can such functions be modeled and implemented?

- AHEAD constants and functions are typed. The layer directories that we have composed with AHEAD have the same structure (or type), but our tools are not yet sophisticated enough to validate this assumption. A theory is needed to type refinements and artifact hierarchies.
- AHEAD has been developed with functions that have at most one parameter. There are GenVoca models with multi-parameter functions [2]. How can such functions be implemented in AHEAD? Why are functions with one parameter so common? We suspect that the answer is not how domains are represented, but rather our (implicit) use of currying which expresses a function of multiple parameters as a composition of functions with single parameters.
- Most, but not all, GenVoca models have focussed on the synthesis of programs using statically composable refinements. However, neither GenVoca or AHEAD preclude dynamic composability. How does AHEAD generalize?

8 Related Work

Among the most advanced work on generators is that of *Model Integrated Computing (MIC)* at Vanderbilt [25]. MIC embraces the concept that architects use multiple representations to specify application designs, and MIC generators have been developed to synthesize *graphical domain-specific languages (GDSLs)* that architects can use to specify their designs. Information that is collected from GDSL specifications are integrated and stored in a database. Specific artifacts of a design, ranging from source code to representations for analysis tools (e.g., model checkers), can be extracted and deduced from this database.

MIC has had great success in synthesizing software and hardware in engineering and manufacturing domains where the building blocks of systems and their composition-by-construction paradigms are well-understood. Where MIC has had less success is in areas of classical software applications where the building blocks and construction-by-composition paradigms are not well understood. We believe this is where AHEAD contributes: AHEAD shows how domains can be modularized as features and how applications of the domain are composed from them.

Domain-specific languages are recognized as a more efficient way in which to (a) specify applications and (b) integrate domain-specific analyses to validate DSL programs [30]. DSL usage is increasing and has been particularly successful in the specification of product-lines [31]. AHEAD not only embraces the use of DSLs as the primary means for specifying artifacts, but also advocates that DSL programs can be refined. (Our refinement of state machines and equation files in AHEAD are prime examples; state machines are expressed in a DSL embedded in Java; equation files are written in a stand-alone DSL). It is this scaling of refinements to code and non-code artifacts that is a distinguishing feature of AHEAD.

Aspect Oriented Programming (AOP) is a program refinement technology [18]. AHEAD essentially uses templates to express refinements. A more sophisticated way is to use special compilers to implement AHEAD functions that perform computations on a collective to determine how that collective is to be modified (or “advised”). Aspects are specifications of refinements, and aspect weavers execute these specifications on input programs. Thus, aspects provide another important kind of AHEAD function that is currently lacking in the AHEAD tool set.

Gray has shown how aspects apply to non-code artifacts [12]. AHEAD shows how both code and non-code artifacts can be refined simultaneously in collection hierarchies.

Multi-Dimensional Separation of Concerns (MDSC) is another program refinement technology [22]. We have built GenVoca generators using `Hyper/J`. Layers correspond to hyperslices, and GenVoca equations correspond to compositions of hyperslices. Further, MDSC advocates that the techniques for assembling customized code from hyperslice compositions should also work for non-code artifacts as well. This conjecture inspired work on AHEAD. The contribution of AHEAD is a simple algebraic model that supports the MDSC thesis.

9 Conclusions

The future of software engineering lies in automation. Integral to this vision is the transformation of application design from an art into a science — a systematized body of knowledge that is organized around principles, ideally expressible as mathematics. As long as application design remains an art, our abilities to automate key tasks of application design will be limited.

Generators are critical to this vision. As application complexity increases, the burdens placed on generators and their ability to synthesize multiple programs and multiple representations increases. The challenge in scaling refinement-based generators is not one of *possibility*, as there are any number of ad hoc ways in which this can be done. Rather, the challenge is to show how scaling can be accomplished in a *principled* manner, so that generators are not just ad hoc collections of tools and an incomprehensible patchwork of techniques. The significance of this point is

clear: generators are a technological statement that the development of software in a domain is understood well enough to be automated. However, we must make the same claim for generators themselves: the complexity of generators must also be controlled and must remain low as application complexity scales, otherwise generator technology will unlikely have wide-spread adoption.

We have presented the AHEAD model, which offers a practical solution to the above problem. The key ideas are (1) to represent the plethora of representations that define a program — both code and non-code — as a containment hierarchy, and to treat containment hierarchies as constants. And (2) express feature refinements as functions that transform containment hierarchies; such refinements encapsulate all the changes that are to be made to the representations of a program when a feature is added.

Doing this, we discovered that application designs have an elegant hierarchical structure that is expressed by nested sets of equations. By imposing uniformity, we (1) eliminate ad hoc complexity as containment hierarchies scale; (2) enable a small number of operators to be used to manipulate AHEAD concepts; and (3) most importantly keep generators based on step-wise refinement simple as the systems they synthesize scale in complexity.

We reviewed an implementation of AHEAD and described our first non-trivial systems constructed by its principles (FSATS and AHEAD tools). We believe AHEAD takes us an important step closer to realizing a broader vision of automation in software development.

Acknowledgements. We thank Jean-Phillipe Martin, Melanie Kail, Mark Esslinger for their contributions to the design of AHEAD tools and model, and Jim Browne, Stan Jarzabek, Dewayne Perry, and Roberto Lopez-Herrejon for their helpful comments.

10 References

[1] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming", in *Software Reusability II*, T.J. Biggerstaff and A.J. Perlis, Eds., Addison-Wesley, 1989.

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

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

[4] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages", *5th Int. Conf. on Software Reuse*, Victoria, Canada, June 1998.

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

[6] D. Batory, et al., "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", *ACM TOSEM,* April 2002.

[7] D. Batory, R.E. Lopez-Herrejon, J-P. Martin, "Generating Product-Lines of Product-Families", *Automated Software Engineering, September 2002*.

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

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

[10] I. Forman and S. Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1999.

[11] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and Mixins". *ACM Principles of Programming Languages*, San Diego, California, 1998, 171-183.

[12] J. Gray, et al. "Handling Crosscutting Constraints in Domain-Specific Modeling", *CACM* October 2001.

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

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

[15] W. Harrison, C. Barton, M. Raghavachari, "Mapping UML Designs to Java", OOPSLA 2000.

[16] A. Hein, M. Schlick, R. Vinga-Martins, "Applying Feature Models in Industrial Settings", *Software Product Line Conference (SPLC1)*, August 2000.

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

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

[19] H. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for Modular Feature Verification", *Conf. Automated Software Engineering*, 2002.

[20] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development", *OOPSLA 1998*, 97-116.

[21] S. McDirmid, M. Flatt, and W.C. Hsieh, "Jiazzi: new-Age Components for Old-Fashioned Java", *OOPSLA 2001.*

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

[23] 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.

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

[25] J. Sztipanovits and G. Karsai, “Generative Programming for Embedded Systems”, *Generative Programming and Component-Based Engineering (GPCE)*, Pittsburgh, October 2002.

[26] K.J. Sullivan, and D. Notkin, “Reconciling Environment Integration and Software Evolution”, *ACM TOSEM*. 1, 3, July, 1992, 229-268.

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

[28] University of Texas Center for Agile Technology, “AHEAD Tool Documentation”, 2002.

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

[30] A.van Deursen and P. Klint, “Little Languages: Little Maintenance?”, *SIGPLAN Workshop on Domain-Spec. Lang.*, 1997.

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