Scaling Step-Wise Refinement

Don Batory, *Member*, *IEEE*, Jacob Neal Sarvela, *Student Member*, *IEEE*, and Axel Rauschmayer, *Student Member*, *IEEE*

Abstract—*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 noncode 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 from equational specifications, refining Java and non-Java artifacts automatically; a task that was accomplished only by ad hoc means previously.

Index Terms—Specification, design notations and documentation, representation, design concepts, methodologies, data abstraction, extensible languages, program synthesis, feature-oriented programming, refinement.

1 Introduction

Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by incrementally adding details [17]. 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) [23].

There are many implementations of feature refinements, each with different names, capabilities, and limitations: layers [2], feature modules [32], [41], [42], metaclasses [20], collaborations [43], [45], subjects [24], aspects [33], and concerns [48]. More general than traditional packages that encapsulate sets of complete classes, a feature refinement can also encapsulate *fragments* of multiple classes. Fig. 1 depicts a package of 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 package 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 Fig. 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.

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 [26]. Each representation encodes different design information and is expressed in its own language, such as a UML diagram, OCL, XML, etc.

The contribution of this paper shows how step-wise refinement scales to the simultaneous synthesis of multiple programs and multiple noncode representations written in different languages. The challenge is not one of possibility as ad hoc ways are used now. Rather, the challenge is to create an algebraic model of application synthesis that treats *all* representations—code and noncode, individual programs, and multiple programs—in a uniform way. By expressing the refinement of system 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 simple algebraic techniques.

We begin with a review of the GenVoca model [2], 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 models, such as Aspect-Oriented Programming [33] and Multidimensional Separation of Concerns [39] and, thus, our results are not GenVoca-specific. We then review a tool set that supports AHEAD. As a demonstration of AHEAD's viability, we have bootstrapped AHEAD tools, generating over 150K LOC of Java (and other noncode artifacts) solely from equational specifications automatically, a task that was accomplished only by ad hoc means previously.

- *D. Batory and J.N. Sarvela are with the Department of Computer Sciences, University of Texas at Austin, Austin, Texas, 78712. E-mail: {batory, sarvela}@cs.utexas.edu.*
- *A. Rauschmayer is with the Institut für Informatik, Ludwig-Maximilians-Universität München, D-80538 Munich, Germany. E-mail: axel@rauschma.de.*

Manuscript received 30 Sept. 2003; revised 28 Jan. 2004; accepted 30 Mar. 2004.
Recommended for acceptance by L. Dillon.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0153-0903.

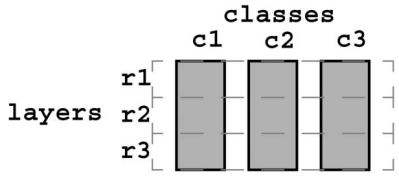

Fig. 1. Classes and refinements (layers).

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 [17] with one major difference: Instead of composing thousands of microscopic program refinements (e.g., `inc(x)` → `x+1`) to yield admittedly small programs, GenVoca scales refinements so that each adds a feature to a program, and composing a few refinements yields an entire program.

2.1 Model Concepts

Initial 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 feature-augmented program as output:

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

A multifeatured 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. A *model* is a set of constants and functions whose members are the building blocks of a product line.

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).[1] It is possible to automatically design software (i.e., find an expression that optimizes some criteria) given a set of declarative constraints for a target application.

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

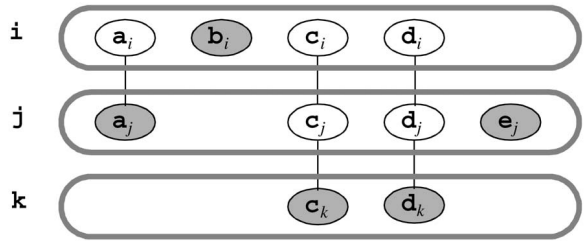

Fig. 2. Implementing refinements by mixin inheritance.

An example of this kind of automated reasoning—historically called *automatic programming* [1]—is [5]. Expression optimization is discussed in more detail in Section 10.1.1.

Although GenVoca constants and functions seem untyped, typing constraints do exist as design rules. *Design rules* capture syntactic and semantic constraints that govern legal compositions. The details of design rules are not germane to our paper, but we do elaborate their importance in Section 10.1.4.

2.2 Model Implementation

A GenVoca constant represents a base program and is a set of classes. A GenVoca function is a set of classes and class refinements. A *class refinement* can introduce new data members, methods, and constructors to a target class, as well as extend or override existing methods and constructors of that class. For years, we emulated class refinement by inheritance using *mixins*—classes whose superclass is specified by a parameter [45], [50]. Mixins can add new data members, methods, and constructors, as well as extend existing methods and constructors of its superclass. Unfortunately, mixins only approximate class refinement because they do *not* inherit constructors of their superclass and a mixin does not assume the name of its superclass. We used simple design techniques to avoid the latter limitation, and illustrate them below.

Fig. 2 depicts the composition `k(j(i))`. Constant or base program `i` encapsulates four classes ($a_i - d_i$). Function `j` refines three classes (vertical lines in Fig. 2 denote class refinement) and adds another class. That is, `j` encapsulates a cross-cut that refines classes a_i, c_i, and d_i (represented by mixins a_j, c_j, and d_j), and adds class e_j. The application of function `k` to `j(i)` results in the refinement of two classes, c_j and d_j.

Linear refinement chains are common in this method of implementation. The composition `k(j(i))` of Fig. 2 produces five classes `a-e`. Each class is synthesized by a linear refinement chain of members from features `i`, `j`, and `k`, which is expressed by the following pseudo-Java code:

```
class a extends aj(ai)        {};
class b extends bi            {};
class c extends ck(cj(ci))    {};
class d extends dk(dj(di))    {};
class e extends ej            {};
```

Above, we represent mixins (class refinements) as functions. That is, mixin $a_j()$ is a function that is applied to base class a_i. The expression $a_j(a_i)$ defines a linear refinement chain of two classes. Class `a` is the name given to

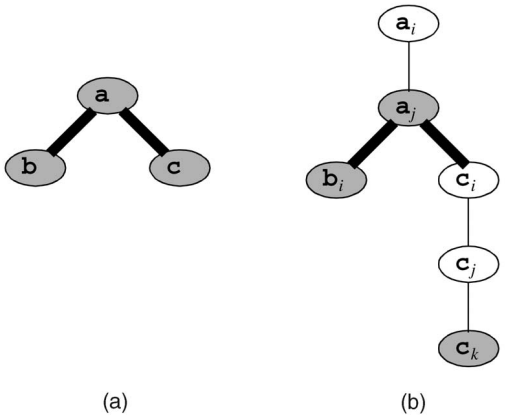

Fig. 3. Refinement of inheritance hierarchies.

the terminal class of this chain. Similarly, $c_k(c_j(c_i))$ defines a linear refinement chain of three classes; class c is the name given to the terminal class of this chain, etc.

The only classes that are instantiated in a synthesized application are the terminal classes of the refinement chains, namely, classes a–e. Nonterminal classes (those that are unshaded in Fig. 2) are *never* instantiated.[2]

Synthesized classes need not be stand-alone; they can belong to an inheritance hierarchy. Suppose classes b and c are subclasses of a (Fig. 3a, where subclassing is shown in bold lines). The refinement chains that we synthesize are the same as that in Fig. 2 except that both classes b_i and c_i are declared to be subclasses of *synthesized class* a. The subtlety here is that inheritance is doing double-duty: It implements subclassing (the bold lines in Fig. 3b) and emulates refinement (relationships shown in thin lines). This design scheme allows us to refine arbitrary subclassing hierarchies by adding new classes and refining existing classes and was the key technique for building the *Jakarta Tool Suite* [4], a set of compiler-compiler tools that use GenVoca models to build product lines of preprocessors for extended Java languages.

Although, arguably, the simplest implementation of class refinements is mixin inheritance [13], [45], [50], more sophisticated means are also possible. Examples include generators [5], [35], program transformations [9], and objects [47].

3 Scaling Refinements

Four ideas have led us to generalize the GenVoca model. First, an application has other representations beyond source code. An application can be defined by UML documents, process diagrams, makefiles, performance models, design rule files, etc., each written in its own language.

This suggests that conventional notions of modularity must be broadened, which leads to our second idea: *A module is a containment hierarchy of artifacts that can include multiple representations of an application*. A traditional example is that of object-oriented source code—a class is a module encapsulating a set of data members and methods, each identified by a *signature*. Similarly, a Java package is another kind of module that encapsulates a set of `.class files`, each identified by its fully qualified class name. A J2EE EAR file is also a module—it encapsulates a set of Java JAR files, deployment descriptors, and HTML files, each identified by its entry name. An application is yet another kind of module which has a representation as a containment hierarchy of named artifacts, including code, makefiles, and documentation.

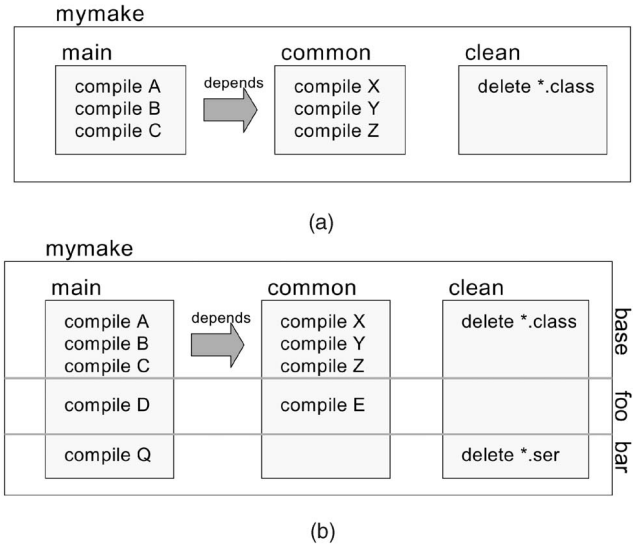

Fig. 4. Makefiles and refined makefiles. (a) Base makefile. (b) A foo and bar refined makefile.

Third, *a scalable notion of refinement should be able to refine all representations in a consistent fashion*. When an application is refined, any or all of its representations may be changed. Thus, if an application is represented by a containment hierarchy, a refinement is a function that transforms arbitrarily nested containment hierarchies, down to the most primitive artifacts. A refinement may alter a containment hierarchy by adding new nodes (e.g., a refined application may introduce new Java or HTML files) or it may refine existing nodes (e.g., existing Java and HTML files are modified).

This leaves us to define refinements for noncode artifacts. The goal is intuitive: Start with an original artifact, apply zero or more refinements to it, and produce an updated artifact. But, what general principle should guide us in defining a representation-specific refinement for an artifact? This leads to our fourth idea, which we call the *Principle of Uniformity*: Impose an object-based structure on artifacts of a given type, taking advantage of any natural indexing scheme that may already exist, and define refinement to follow the notions of mixin inheritance (or more specifically, class refinement).

Consider a makefile, a typical noncode artifact. Fig. 4a 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. Fig. 4b shows the refinement of `base` by `foo`,

2. Smaragdakis and Batory [45] showed cases where instantiation of nonterminal classes might be useful, however, this is possible only using a special casting capability of C++ which may not present in other languages. Our Java implementations, for example, could never instantiate nonterminal classes of a refinement chain.

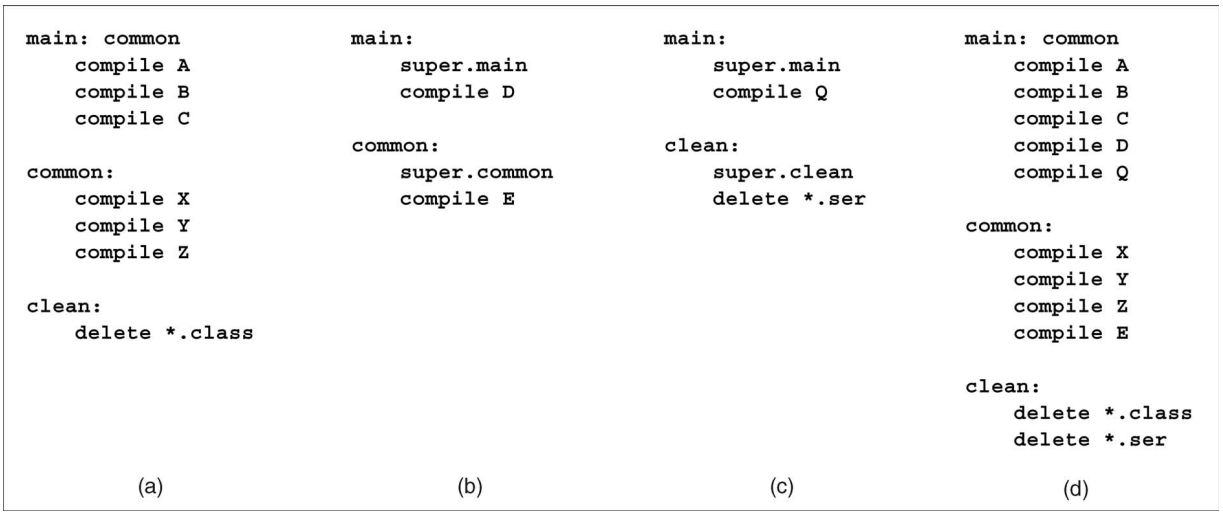

Fig. 5. Makefiles, refinements, and composition. (a) `base makefile`, (b) `foo refinement`, (c) `bar refinement`, (d) `bar(foo(base))`.

which encapsulates a cross-cut of targets that adds a file `D` to compile in `main` and a file `E` to compile in `common`. Fig. 4b also 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`.

A makefile has a natural indexing scheme: Targets are uniquely named (or can be) and each target has associated actions. This is analogous to the association of class method signatures to method definitions. We take advantage of this analogy to impose an object-based structure on makefiles wherein target names play the role of method signatures and target build actions play the role of method definitions. The same idea applies to other makefile concepts, such as properties.

The next steps are to augment the makefile syntax to declare makefile refinements and to create a tool that implements makefile composition. Fig. 5a is a base makefile. Fig. 5b and Fig. 5c, respectively, show one way in which makefile syntax can be augmented—we introduce a keyword, `super`, to specify references from a refining makefile to a target makefile. A makefile composition tool can then expand the `super` references by textual substitution, passing the final result (Fig. 5d) to a make program.

With this approach, it is possible to implement before, after, and overriding advice of "makefile methods" by makefile refinement. New targets and dependencies can also be specified with an appropriate makefile refinement tool. For example, Fig. 6a shows another makefile refinement, `baz`, that overrides the target `common`, augments target `clean` and adds two targets, `gui` and `lnk`, along with appropriate dependencies. The composition `baz(base)`, after textual substitution of `super` references, is shown in Fig. 6b.

Interestingly, many artifact types in use today (XML, HTML, Word documents, etc.) have or can have an object-based structure. However, these types aren't often object-oriented—few support inheritance or more general refinement relationships among their instances. There is, for example, no support for mixin inheritance between Word documents. Nevertheless, the Principle of Uniformity is clear: When introducing a new artifact type, the tasks to be done are to support inheritance relationships between instances and to implement a refinement operation that realizes mixin inheritance. With Word documents, for example, it is reasonable to support inheritance by building on a natural indexing scheme for chapters, sections, and paragraphs and—admittedly, a daunting task—to implement a refinement operation by textual substitution.

Given the ability to refine noncode artifacts, we can now 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

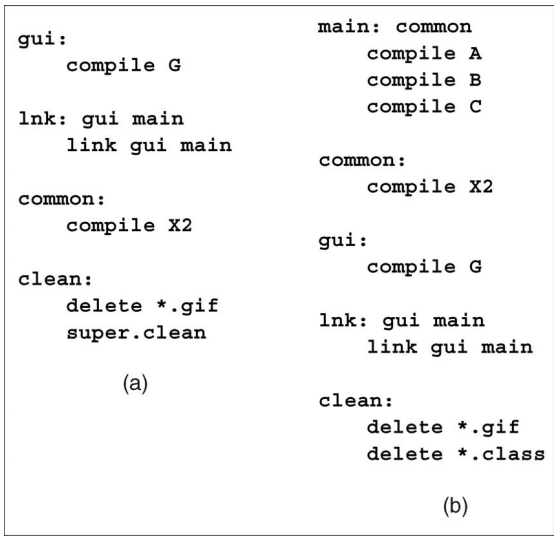

Fig. 6. Makefile refinement and composition. (a) `baz refinement`, (b) `baz(base)`.

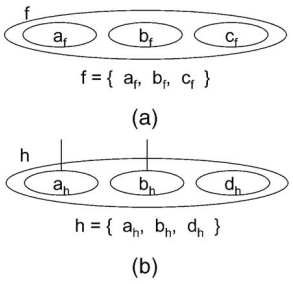

Fig. 7. Constants and functions as sets (collectives). (a) Constant f is a set of constraints. (b) Function h is a set of functions and constants.

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 as a series of functions (refinements) applied to a constant (base artifact).

Fig. 7a depicts our graphical notation for a GenVoca constant f that encapsulates base *artifacts* $\mathtt{a}_f - \mathtt{c}_f$. Instead of pictures, we express constant f mathematically as a set of constants: $\mathtt{f} = \{\mathtt{a}_f, \mathtt{b}_f, \mathtt{c}_f\}$. Similarly, Fig. 7b depicts our graphical notation for a GenVoca function h that encapsulates functions $\mathtt{a}_h$ and $\mathtt{b}_h$ and constant $\mathtt{d}_h$. Function h can be expressed mathematically as a set of functions and constants: $\mathtt{h} = \{\mathtt{a}_h, \mathtt{b}_h, \mathtt{d}_h\}$.

4.2 Composition

We use the term *collective* as an alternative to "set." Members of a collective are units that represent constants or functions. Instead of writing h(f) to denote the composition of h with f, we henceforth write h•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 inner term:

$$\begin{aligned}\mathtt{h} \bullet \mathtt{f} &= \{\mathtt{a}_h, \mathtt{b}_h, \mathtt{c}_h\} \bullet \{\mathtt{a}_f, \mathtt{b}_f, \mathtt{d}_f\} \\ &= \{\mathtt{a}_h \bullet \mathtt{a}_f, \mathtt{b}_h \bullet \mathtt{b}_f, \mathtt{c}_h, \mathtt{d}_f\}.\end{aligned}$$

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

$$\begin{aligned}&\mathtt{h} \bullet \mathtt{f} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\} \\ &\texttt{where } \mathtt{a} = \mathtt{a}_h \bullet \mathtt{a}_f \\ &\qquad\quad \mathtt{b} = \mathtt{b}_h \bullet \mathtt{b}_f \\ &\qquad\quad \mathtt{c} = \mathtt{c}_h \\ &\qquad\quad \mathtt{d} = \mathtt{d}_f\end{aligned}$$

Each expression in a collective defines a refinement chain of an artifact that is to be produced. Fig. 8 shows this correspondence between our graphical notation and its AHEAD expression: $\mathtt{a}_h \bullet \mathtt{a}_f$ is the refinement chain for artifact a and $\mathtt{b}_h \bullet \mathtt{b}_f$ is the chain for b. Artifacts c and d are not refined, so they are unchanged from their original definitions.

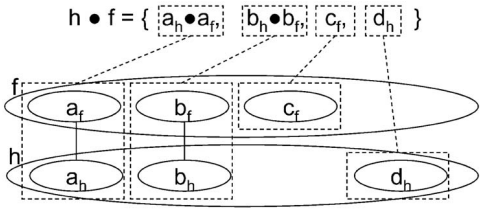

Fig. 8. Expression and refinement chain correspondence.

4.2.1 Containment Hierarchies

Compound artifacts are expressed by nesting: Units may themselves be collectives. Composition of compound artifacts is achieved by composing nested collectives. For example, suppose $\mathtt{a}_h$ and $\mathtt{a}_f$ in Fig. 8 are the collectives $\mathtt{a}_h = \{\mathtt{x}_h, \mathtt{z}_h\}$ and $\mathtt{a}_f = \{\mathtt{x}_f, \mathtt{y}_f\}$. The expression $\mathtt{a}_h \bullet \mathtt{a}_f$ expands to the collective $\{\mathtt{x}_h \bullet \mathtt{x}_f, \mathtt{y}_f, \mathtt{z}_h\}$. The maximum 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, etc.

A compound artifact is a containment hierarchy. A Java program p, for example, is a compound artifact and, thus, is a nonleaf node; its leaves might be the set of Java files {x.java, y.java,...} that implement it, the set of XML files {g.xml, h.xml,...} that defines p's build scripts, etc.

Thus, a feature—whether it represents an AHEAD constant or a function—is defined by a tree of units. When features are composed, all of their corresponding units are composed. Thus, feature composition has a simple interpretation.

4.2.2 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 artifact-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.

4.2.3 Scalability

OO programming languages that support parameterized inheritance can define and refine code artifacts, but are unsuitable for noncode representations. In contrast, equations elegantly express refinement relationships for *all* representations. Furthermore, *equations and containment hierarchies 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

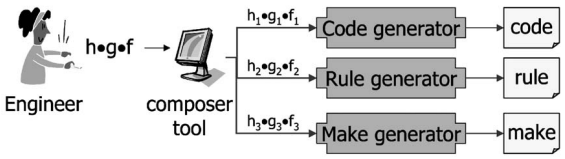

Fig. 9. Organization of AHEAD generators.

equation into its constituent artifact equations. Thus, a simple `composer` tool does the work of orchestrating other relatively simple artifact-type-specific tools to produce the complex set of artifacts that comprise a synthesized system (Fig. 9).

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). Fig. 10a shows a collective that defines feature `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}`. Fig. 10b depicts its representation as a directory.

Feature composition is directory composition. A composite directory is produced whose organization is isomorphic to the directories from which it was composed. Fig. 10c shows a composition of features `A` and `B` resulting in feature `C`. Corresponding units in each directory are composed to produce a compound unit. For example, `X.jak` of `C` is the composition of `X.jak` of `B` with `X.jak` of `A`.

Given this organization, we initially built AHEAD tools using the Jakarta Tool Suite mentioned earlier [4]. Almost all AHEAD tools were written in a JTS-produced dialect of Java that includes refinement declarations, metaprogramming constructs (e.g., Lisp "quote" and "unquote"), and hygienic macros. We have since bootstrapped our implementation of AHEAD tools. We discuss this topic further in Section 8.

The main tool of AHEAD is the `composer`, which takes an equation as 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 because verifying the code synthesis capabilities of AHEAD was our first priority. Subsequently, composition tools for noncode artifacts, such as HTML files, makefiles, equation files, design rule files, XML files, and BNF-grammar files (for synthesizing extensible preprocessors) were constructed. We anticipate the set of composition tools will grow larger over time. In the following sections, we illustrate composable code and noncode artifacts that are supported by AHEAD.

5.1 Code Artifacts: Jak Source

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 *domain-specific languages* (*DSLs*) for refinements, state machines, and metaprogramming. 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 produces 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 (Fig. 11).

5.1.1 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 `feature` declaration which specifies the name of the feature to which the file belongs (see Fig. 12a). More interesting is a state machine declaration, which consists of state and edge (transition) declarations. The state machine `fsm` of Fig. 12b declares three states (`s1-s3`) and two edges (`e1-e2`). See [6] for more details.

A code function refines an interface, class, or state machine. The `refines` modifier distinguishes constant declarations from functions. A refinement of class `k` in Fig. 12a is shown in Fig. 12c. It adds a new data member

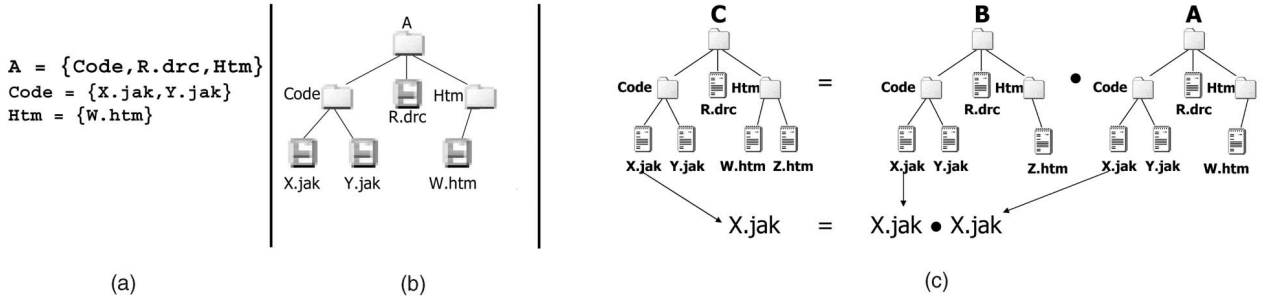

Fig. 10. Collectives as directories. (a) A collective, (b) its directory, and (c) directory composition.

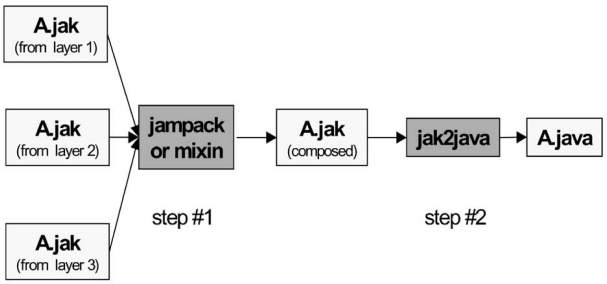

Fig. 11. Composing and translating Jak files.

(counter2) and a new method (method2). Methods can be refined/extended in the usual way by overriding the original method and invoking that method via a super call in the extension body. A refinement of state machine fsm is shown in Fig. 12d. It adds state s4 and edge e3.

5.1.2 Composition

jampack and mixin are sophisticated tools, both over 30K Java LOC in size. Their functionality, however, is fairly simple. Conceptually, 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 Fig. 12 is shown in Fig. 13a; 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 Fig. 12 is shown in Fig. 13b. 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:

1. compose Jak files,
2. translate the composite Jak file to its Java counterpart,
3. compile and debug the Java file, and
4. 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 feature boundaries, thus changes made to the composite Jak file must be manually propagated back to the original Jak feature files. This can be tedious and error prone.

mixin was created as an alternative implementation of the Jak composition operator. It preserves feature boundaries by creating an inheritance hierarchy—refinement chain—of class/interface/state machine declarations within a single Jak file, where the bottom-most declaration is public and all others are abstract. Every class/interface/state machine declaration is prefaced by a SoUrCe statement that identifies both the feature name and path to the file from which that instantiated declaration originated.[3]

3. Adding new reserved words to a language may preclude the compilation of legacy programs as the new word was previously legal as a program identifier. The case alteration in keyword "SoUrCe" minimizes the likelihood of conflict with legacy program identifiers.

Figs. 14a and 14b shows the mixin output for the corresponding jampack compositions of Figs. 13a and 13b. 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 feature files. Using unmixin saves a considerable time during a development cycle.

There is yet another strategy: It is possible to edit the original feature files and recompose. Our experience suggests that debugging features 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 Noncode Artifacts: Grammars

Grammars are defined in AHEAD by a BNF-syntax. A grammar for an elementary calculator that supports only integer summations is shown in Fig. 15a. The INTEGER token is predefined and the PLUS token is explicitly defined.

A refinement of a base grammar is a grammar fragment that defines 1) new tokens, 2) new productions, and 3) extensions to previously defined productions. A refinement of the calculator grammar that generalizes computations to additions and subtractions is shown in Fig. 15b. It defines a new token (MINUS) and extends a previously defined production (Operator). The "Operator : super" construct says extend an existing Operator production.

The composition of Fig. 15a and Fig. 15b is Fig. 15c, which is the union of the token definitions and the resulting set of productions. Grammar refinement follows the Principle of Uniformity by equating tokens with data members and productions with methods. Grammars can be refined with the addition of new data members (tokens), new methods (productions), and extensions of previously defined methods (productions).

Grammar composition arises in the synthesis of AHEAD tools because the tools are language extensible. When a feature extends a base language (e.g., Java), we compose grammars of the base language and its extension to produce the composite grammar. From this grammar, we can synthesize parsers and other analysis programs using standard compiler-compiler tools.

5.3 Noncode Artifacts: Equation Specifications

An *equation file* encodes an AHEAD equation, which is a specification for synthesizing a code or noncode artifact. The file for X = F•E•D is depicted in Fig. 16a; the name of the file is X.equation and the file itself is a text file that lists one feature per line, innermost feature first. A refinement of an equation follows the principle of uniformity: We treat an equation as a method and a refinement is an equation that may reference its parent equation using super. A refinement of an equation is depicted in Fig. 16b, and the result of composing the equation files of Figs. 16a and 16b is Fig. 16c. If a refinement does not reference super, it is a constant and overrides the parent definition.

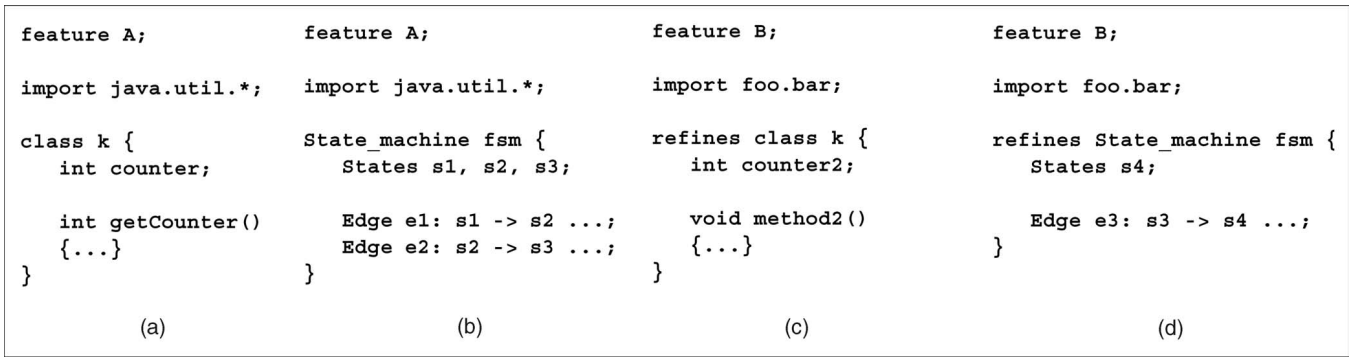

Fig. 12. AHEAD code constants and functions. (a) Class constant, (b) state machine constant, (c) class function, (d) state machine function.

Depending on the replacement of `super` in an equation file, a refinement might add new features before, after, or around the original equation file (like before, after, and around methods [33]).

Equation files are useful as command-line input to `composer` and for implementing metamodels, the topic of Section 7.

5.4 Additional Noncode Artifacts

We use a special domain-specific language for defining the design rules. These rules, essentially preconditions and postconditions, define the legal ordering in which units can be composed. The composition of design rule preconditions and postconditions follows standard techniques [54] and is described in detail in earlier research [3], [40] and AHEAD documentation [49].

AHEAD expressions can be optimized in the same way as that relational algebra expressions in database systems can be optimized [5]. Each operator (constant or function) requires three distinct representations: a code representation to implement the operator, a design rule representation for composition validation, and a performance model representation to evaluate the efficiency of a composition. See [5] for details on how performance models are represented and refined and how equations were optimized.

6 Examples: Calculator Models

Three elementary AHEAD models are sketched in detail in this section. Beyond their expositional value, they serve double-duty to illustrate that different AHEAD models can have relationships and these relationships can be expressed by more abstract AHEAD models called metamodels, the topic of Section 7.

6.1 `calc` Model

Consider the synthesis of a family of postfix calculators, where features are arithmetic operations. An AHEAD model would have a single constant, $\texttt{base}_{calc}$, representing a calculator with no operations. Refinements add individual operations—addition (`add`), subtraction (`sub`), etc.—to the calculator:

$$\texttt{calc} = \{\texttt{base}_{calc}, \texttt{add}_{calc}, \texttt{sub}_{calc}, \ldots\}.$$

Consider the units of `calc`. Each unit is a collective with a single file:

$$\texttt{base}_{calc} = \{\texttt{cal}_{base}\texttt{.jak}\},$$
$$\texttt{add}_{calc} = \{\texttt{cal}_{add}\texttt{.jak}\},$$
$$\texttt{sub}_{sub} = \{\texttt{cal}_{sub}\texttt{.jak}\}.$$

The source for these files is listed in Fig. 17. $\texttt{cal}_{base}\texttt{.jak}$ is the base calculator, $\texttt{cal}_{add}\texttt{.jak}$ introduces the `add()` operation, and $\texttt{cal}_{sub}\texttt{.jak}$ introduces the `sub()` operation. Note that both $\texttt{cal}_{add}\texttt{.jak}$ and $\texttt{cal}_{sub}\texttt{.jak}$ are class refinements as they add a method to class `cal`.

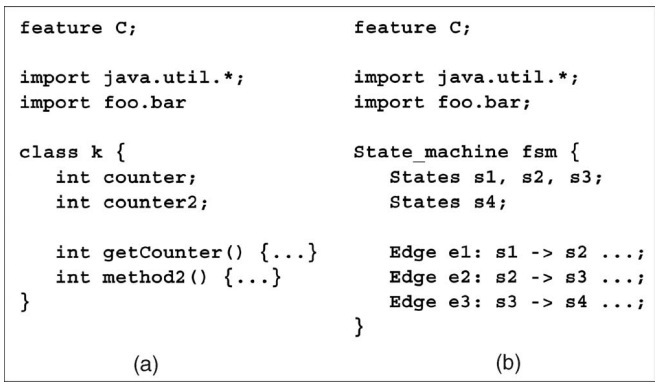

Fig. 13. `jampack` compositions of Jak files.

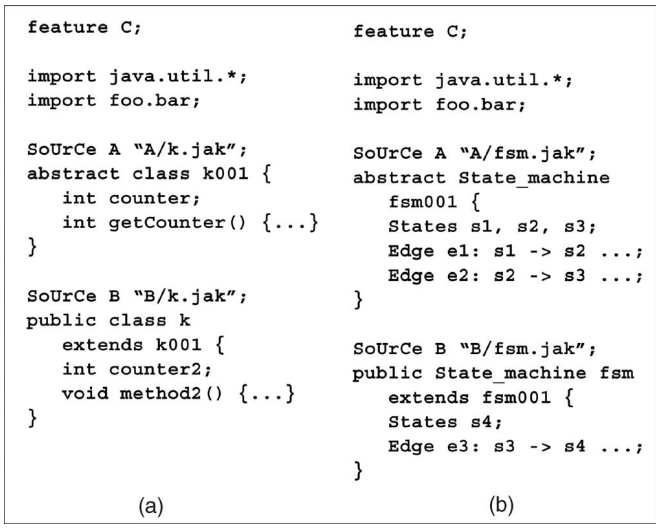

Fig. 14. Mixin compositions of Jak files.

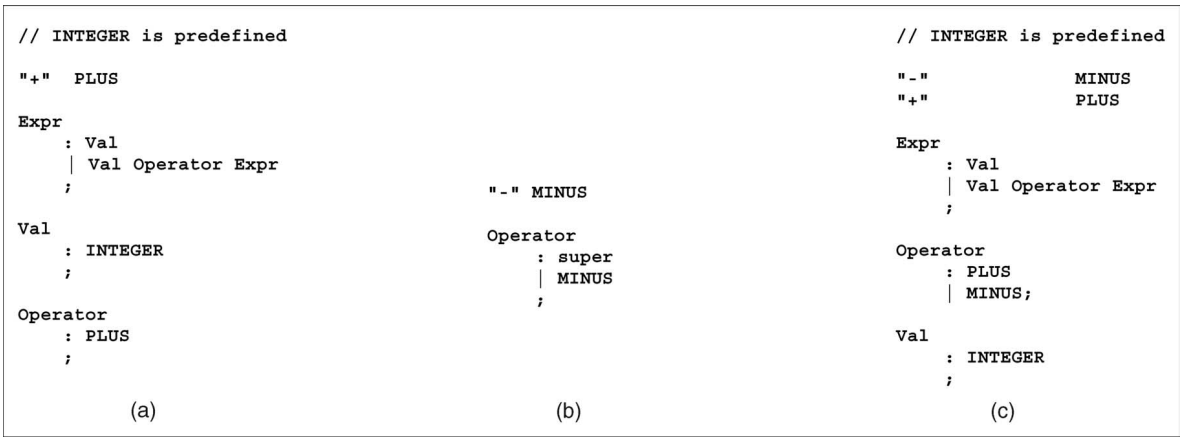

Fig. 15. Grammar files: (a) constant, (b) function, and (c) composition.

To synthesize a calculator `mycalc` that has `add` and `sub` operations, we evaluate the equation:

$$\texttt{mycalc} = \texttt{sub}_{calc} \bullet \texttt{add}_{calc} \bullet \texttt{base}_{calc}.$$

The result is Fig. 18. Because there are no method refinements, the target class `cal` is simply the union of the members in Figs. 17a, 17b, and 17c. Note that the order in which $\texttt{add}_{calc}$ and $\texttt{sub}_{calc}$ are composed does not matter as they are independent of each other. That is, the order in which the `add()` and `sub()` methods are added to the cal class is inconsequential.

6.2 `gui` and `cl` Models

Now, suppose we want to synthesize a client-callable interface for a calculator. A model that synthesizes a GUI for a calculator is `gui`, whose units are in 1-to-1 correspondence with `calc` units:

$$\texttt{gui} = \{\texttt{base}_{gui}, \texttt{add}_{gui}, \texttt{sub}_{gui}\}$$
$$\texttt{where}\ \ \texttt{base}_{gui} = \{\texttt{gcal}_{base}\texttt{.jak}\}$$
$$\texttt{add}_{gui} = \{\texttt{gcal}_{add}\texttt{.jak}\}$$
$$\texttt{sub}_{gui} = \{\texttt{gcal}_{sub}\texttt{.jak}\}.$$

Like `calc`, each unit of `gui` is a collective with a single file. The source for these files is listed in Fig. 20. Briefly, file $\texttt{gcal}_{base}$`.jak` defines a base GUI class that has a text field and Clear and Enter buttons. The GUI that is displayed by $\texttt{gcal}_{base}$`.jak` is shown in Fig. 19. The key methods of this class are:

- `initAtoms()`—where GUI atoms (buttons, text fields) are initialized,
- `initLayout()`—where GUI atoms are linked into a layout containment hierarchy for display, and
- `initListeners()`—where event listeners and event actions are coded.

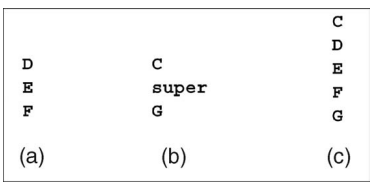

Fig. 16. Equation Files: (a) constant, (b) function, and (c) composition.

When a calculator operation is added to a GUI (such as the `add` and `sub` operations), the $\texttt{gcal}_{base}$`.jak` class is refined in the following way:

- A GUI button is declared for the operation (this adds a new data member to the `gcal` class).
- The button is initialized by extending the `initAtoms()` method.
- The button is added to the containment hierarchy by extending the `initLayout()` method.
- Button-click events and event actions are added to the `initListeners()` method.

When we evaluate the equation `mycalc` using the `gui` model:

$$\texttt{mycalc} = \texttt{sub}_{gui} \bullet \texttt{add}_{gui} \bullet \texttt{base}_{gui},$$

we synthesize a single class `gcal` that displays the GUI in Fig. 21. In general, each unit of the `gui` model that corresponds to a calculator operation adds a new button to the bottom row of this figure. If n operation units are composed, there will be n buttons, one per operation in the GUI.

By the same reasoning, it is possible to imagine a `cl` model (one that gives a command-line interface to a calculator) that works along the same lines:

$$\texttt{cl} = \{\texttt{base}_{cl}, \texttt{add}_{cl}, \texttt{sub}_{cl}, \ldots\}.$$

Composing these units synthesizes a class that has a `main` method with a command-line parser that understands how to parse and execute calls to designated operations.

6.2.1 Perspective

The `calc`, `gui`, and `cl` models can be loosely coupled. For example, there may already exist a fully configured calculator class and the `gui` and `cl` models allow us to synthesize a customized interface to this class that exposes only selected operations. Alternatively, these models could be tightly coupled so that the calculator *and* its user-callable interfaces could be synthesized in lock-step. Tightly coupled models are common and are formalized by metamodels.

```
import java.math.BigDecimal;

class cal {
   static BigDecimal zero = new BigDecimal( "0" );

   // e0,e1,e2 emulate a 3-element stack
   BigDecimal e0 = zero, e1 = zero, e2 = zero;

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

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

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

(a)

```
refines class cal {
    void add() {
       e0 = e1.add( e0 );
       e1 = e2;
    }
}
```

(b)

```
refines class cal {
    void sub() {
        e0 = e1.subtract( e0 );
        e1 = e2;
    }
}
```

(c)

Fig. 17. Calculator source code: (a) $\texttt{cal}_{base}\texttt{.jak}$, (b) $\texttt{cal}_{add}\texttt{.jak}$, and (c) $\texttt{cal}_{sub}\texttt{.jak}$.

7 Advanced Topic: Metamodels

Recall that a model is a collective whose units are building blocks of a product line. Suppose model M0 is {a,b,c,d}. Further, suppose there are variations of M0, such as:

```
M1 = { a, b, e }
M2 = { c, d, e }
```

We can define an AHEAD model, called a *metamodel*, that expresses a *product line* of models. A *metamodel* whose product line includes M0-M2 is MM:

```
MM = { AB, CD, E }
where  AB = { a, b }
       CD = { c, d }
       E = { e }
```

Given MM, we can synthesize the models M0-M2 by composing units of MM:

```
M0 = AB•CD
M1 = AB•E
M2 = CD•E
```

Thus, the units of MM are building blocks of models. The following are interesting uses of metamodels.

```
import java.math.BigDecimal;
class cal {
   static BigDecimal zero = new BigDecimal( "0" );
   BigDecimal e0 = zero, e1 = zero, e2 = zero;

   void add() {
      e0 = e1.add( e0 );
      e1 = e2;
   }
   void clear() {
      e0 = e1 = e2 = zero;
   }
   void enter( String val ) {
      e2 = e1;
      e1 = e0;
      e0 = new BigDecimal( val );
   }
    void sub() {
      e0 = e1.subtract( e0 );
      e1 = e2;
   }
   String top() {
      return e0.toString();
   }
}
```

Fig. 18. Synthesized cal.jak file.

Fig. 19. GUI base.

7.1 Service Packs

A *service pack* is an update to a model. A *service pack metamodel* SP contains an initial model $\texttt{M}_0$ and a series of service pack updates $\texttt{S}_1$, $\texttt{S}_2$, $\texttt{S}_3$, each of which incrementally updates a model:

$$\texttt{SP} = \{\texttt{M}_0, \texttt{S}_1, \texttt{S}_2, \texttt{S}_3, \ldots\}.$$

A composition operator $\diamond$, called *replace*, is used to apply the changes of a service pack to an existing model. If $\texttt{U}_k$ and $\texttt{U}_j$ are primitive (i.e., noncompound) units, the law of the replace operator is:

$$\texttt{U}_k \diamond \texttt{U}_j = \texttt{U}_k \quad // \ \texttt{U}_k \ \texttt{replaces} \ \texttt{U}_j.$$

Otherwise, $\diamond$ is identical to the composition $\bullet$ operator for collectives. ($\diamond$ is actually a special case of $\bullet$). Thus, a model M that is up-to-date with regard to service pack $\texttt{S}_3$ is defined by the equation:[4]

$$\texttt{M} = \texttt{S}_3 \diamond \texttt{S}_2 \diamond \texttt{S}_1 \diamond \texttt{M}_0.$$

That is, the effects of $\texttt{S}_1$ are applied to $\texttt{M}_0$ by replacing old base artifacts with new ones and adding new artifacts. The same for $\texttt{S}_2$ and $\texttt{S}_3$. Special primitive units might be used to indicate the physical deletion (rather than replacement) of designated files.

7.2 A Calculator Metamodel

Let's return to the calc, gui, and cl models of Section 6. Because the units of these models are in 1-to-1 correspondence, a tight coupling between these models is to embed them as units of a metamodel CMM:

4. A design rule is: $\texttt{S}_i$ must be applied before $\texttt{S}_{i+1}$ for $i \geq 0$.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class gcal extends JFrame {

    gcal( String AppTitle ) {
        super( AppTitle );
        initAtoms();
        initLayout();
        initContentPane();
        getContentPane().add( ContentPane );
        initListeners();
        pack();
        setVisible( true );
    }

    // declare and initialize atomic components
    cal calc;
    JTextField Result;
    JButton clear, enter;

    public void initAtoms() {
        calculator = new cal();
        Result = new JTextField( 10 );
        clear = new JButton( "Clear" );
        enter = new JButton( "Enter" );
    }

    // declare and initialize layout components here
    JPanel clearEnter;
    JPanel ops;

    public void initLayout() {
        clearEnter = new JPanel();
        clearEnter.add( clear );
        clearEnter.add( enter );
        ops = new JPanel();
    }

    // initialize ContentPane here
    JPanel ContentPane;

    public void initContentPane() {
        ContentPane = new JPanel();
        ContentPane.setLayout( new GridLayout( 0,1 ) );
        ContentPane.setBorder(BorderFactory.createEtchedBorder());
        ContentPane.add( Result );
        ContentPane.add( clearEnter );
        ContentPane.add( ops );
    }

    // initialize listeners here
    public void initListeners() {
         clear.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                calc.clear();
                Result.setText( "" );
            }
        } );
        enter.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                calc.enter( Result.getText() );
                Result.setText( calculator.top() );
            }
        } );
     }
     public static void main( String[] args ) {
        new gcal( "gcal" );
     }
}
```

(a)

```
refines class gcal{
    JButton plus;

    public void initAtoms() {
      super.initAtoms();
      plus = new JButton( "+" );
    }

    public void initLayout() {
      super.initLayout();
      ops.add( plus );
    }
    public void initListeners() {
       super.initListeners();
        plus.addActionListener(
          new ActionListener() {
          public void
            actionPerformed( ActionEvent e ) {
              calc.enter(Result.getText() );
              calc.add();
              Result.setText( calc.top() );
          }} );
    }
}
```

(b)

```
refines class gcal{
    JButton sub;

    public void initAtoms() {
      super.initAtoms();
      sub = new JButton( "-" );
    }

    public void initLayout() {
      super.initLayout();
      ops.add( sub );
    }

    public void initListeners() {
       super.initListeners();
       plus.addActionListener(
         new ActionListener() {
          public void
            actionPerformed( ActionEvent e ) {
              calc.enter( Result.getText() );
              calc.sub();
              Result.setText( calc.top() );
         }} );
    }
}
```

(c)

Fig. 20. Calculator GUI source code. (a) $\texttt{gcal}_{base}\texttt{.jak}$, (b) $\texttt{gcal}_{add}\texttt{.jak}$, (c) $\texttt{gcal}_{sub}\texttt{.jak}$

Fig. 21. Refined GUI.

CMM = { calc, gui, cl }.

calc is the (lone) constant and gui and cl are functions that add different front ends. The product line of CMM has four distinct models:

```
basic = calc        // no front end
wgui = gui•calc     // with gui front end
wcl = cl•calc       // with command-line
                       front end
```

Fig. 22. A declarative feature specification.

```
clg = cl•gui•calc   // with both command-line
                       and gui front ends
```

Consider wgui. It is a model that allows us to *simultaneously synthesize* a calculator class *and* its corresponding GUI front end. Look at the contents of wgui by expanding its definition:

$$\begin{aligned}
\texttt{Wgui} &= \texttt{gui} \bullet \texttt{calc} \\
&= \{\texttt{base}_{wgui}, \texttt{add}_{wgui}, \texttt{sub}_{wgui}\} \\
\text{where}\ \ \texttt{base}_{wgui} &= \texttt{base}_{gui} \bullet \texttt{base}_{calc} \\
&= \{\texttt{gcal}_{base}\texttt{.jak}, \texttt{cal}_{base}\texttt{.jak}\} \\
\texttt{add}_{wgui} &= \texttt{add}_{gui} \bullet \texttt{add}_{calc} \\
&= \{\texttt{gcal}_{add}\texttt{.jak}, \texttt{cal}_{add}\texttt{.jak}\} \\
\texttt{sub}_{wgui} &= \texttt{sub}_{gui} \bullet \texttt{sub}_{calc} \\
&= \{\ \texttt{gcal}_{sub}\texttt{.jak}, \texttt{cal}_{sub}\texttt{.jak}\}.
\end{aligned}$$

That is, the wgui model has the same number of units with the same names (ignoring subscripts) as calc and gui, namely, $\{\texttt{base}_{wgui}, \texttt{add}_{wgui}, \texttt{sub}_{wgui}\}$. Further, each unit encapsulates a pair of files: One file represents the calculator base or refinement, the second represents the corresponding GUI base or refinement. For example, $\texttt{sub}_{wgui}$ encapsulates the refinement ($\texttt{cal}_{sub}$.jak) that adds the sub() operation to the base calculator class and another refinement ($\texttt{gcal}_{sub}$.jak) that integrates the subtraction button into the base GUI class.

The significance of metamodels like CMM is that it *separates concerns*: It separates a design specification from its possible implementations. For example, the design of the mycalc application is specified by the equation mycalc = sub•add•base. This equation states that mycalc has addition and subtraction operations *without specifying their implementation*. To supply an implementation, mycalc must be evaluated in the context of a model. If the model is calc, then only the calculator source is generated. If the model is wgui, then both the calculator source *and* its GUI are generated. *In this example, metamodels separate the specification of an application's design from a family of implementations*. Further, they allow us to create declarative feature-oriented specifications expressed as GUI front ends like Fig. 22. The check-boxes in the left-most column of the GUI specify the features of a calculator's design; the check-boxes in the right column specify the model to use. From this input, the mycalc equation and metamodel equation cgl can be inferred. Together, these equations specify a particular calculator in a large space of possible calculators and implementations.

Making feature design specifications orthogonal to feature implementation specifications leads to an interesting result. Instead of specifying an application by a *single* equation, we are using a *pair* of equations to define an application. We generalize this approach in [7], [8] so that a single application is specified by n equations, one for each orthogonal concern. We call this generalization *Origami*; it's significance is scalabililty. We show that the length of each of the n equations will be at most k terms. (Our calculator metamodel is a particular example of Origami where $n = 2$ and $k = 3$). Thus, the length of an Origami specification is $O(kn)$. Had we not used Origami (or metamodels), we show that the corresponding AHEAD specification could be exponentially longer—$O(k^n)$ terms. More examples of Origami are given in [7], [8], where we explain how AHEAD tools are synthesized from compact Origami equations.

8 Applications

AHEAD is being used to build next-generation distributed *fire support simulators* (*FSATS*) for the US Army *Simulation, Training, and Instrumentation Command* (*STRICOM*). Several years ago, we built a feature-oriented prototype of FSATS [6]. As the first nontrivial test of AHEAD, we converted the prototype's source code, which included classes, interfaces, and state machines, into AHEAD features. We also added design rule files and makefiles, so, when AHEAD features were composed, three very different representations of FSATS (code, design rules, makefiles) were synthesized. The prototype was defined by a single AHEAD equation composing 21 features, yielding 90 files and 4,500 Java LOC.

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 features. In addition to code representations, AHEAD directories included grammar (BNF) files, which defined the syntax of optional Java language extensions. 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], [8].

To convey the complexity of this bootstrapping step, we offer three arguments:

- There are 69 distinct AHEAD features constituting a code base of 33K Jak LOC. The AHEAD equation for each tool references approximately 23 features, where 10 features are shared among AHEAD tools. Each tool is generated by composing the Jak and grammar representations of each feature and translating their representations to Java, resulting in over 30K LOC per tool. Thus, using only equational specifications, we are generating the AHEAD tool suite (150K Java LOC [7]) *automatically*, a task that was accomplished only by ad hoc means previously.
- We subjectively estimate that to write the main tools of ATS by hand would take at least two man-years. The effort to manually replicate the entire tool suite, which goes beyond the main tools to include code formatters, document generators, a graphical interface, etc., could be three or more times that. We cannot experimentally validate these estimates as it would be too costly. But, more importantly, *to make these tools feature-extensible—the prime motivation for*

AHEAD—would require a toolkit like AHEAD to be available in the first place.

- Another reason to bootstrap AHEAD is to demonstrate feature extensibility on a nontrivial application. Bootstrapping a compiler is a major milestone in compiler development and it is a convincing test of a compiler to compile itself. In compilers, there are multiple levels of abstraction that are superimposed in the code. Bootstrapping requires these levels to be separated. Doing so is nontrivial. The same holds for ATS, but it is *more* difficult because ATS is not just a single tool, but a suite of interdependent tools each requiring different representations to be composed.

In short, bootstrapping ATS was a major achievement. We believe ATS is the most sophisticated system ever produced by automatic step-wise refinement.

9 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—see [34].
- There are *many* operators on collectives and units besides the • and ⋄ operators. `javac`, for example, could be an operator on a collective that compiles the encapsulated Java files to another collective. `javaDoc` could derive HTML documentation from the same Java files and, in general, many development tools can be realized as operators on collectives. By equating standard tools with operators, we enrich our algebra for software development. Once applications are specified algebraically, they are amenable to automated optimization and reasoning.
- 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 in a refinement (e.g., English explanations of source code) is something that we currently accomplish manually. Tool support for artifact derivation and consistency maintenance among artifacts is needed.
- Many refinements that impact—cross-cut—different parts of programs require more sophisticated implementations of refinements than used in AHEAD. For example, 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 feature 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 type theory is needed to express refinements and artifact hierarchies [15].
- In Section 4.2, we mentioned that the • operator was polymorphic. Our current implementation realizes • by overloading. Even though • appears to be higher-order, AHEAD implements composition as precompile-time static transformations which treat • operands purely as typed data, where every type is a subtype of Unit. When • is applied to Java code, a Java-specific transformation tool is invoked and, in general, a type-specific tool is invoked to implement composition. In this sense, • is bounded polymorphic. Again, a type theory for AHEAD would further clarify these issues.
- Our code composition tools are preprocessors. What is eventually needed are proper extensions to programming languages that allow separate compilations of features. Our concept of code refinement is similar to the open class concept in MultiJava [16] where separate compilation and type checking of class extensions (which could correspond to AHEAD features) are correctly handled, but MultiJava does not support method refinement or the composition of selected extensions.
- AHEAD has been developed with functions that have at most one parameter. There are, however, GenVoca models containing functions with multiple parameters [2]. Are there cases where multiple-parameter functions are more natural to software developers? If so, how can such functions be implemented in AHEAD? This issue is analogous to that of single inheritance versus multiple inheritance.
- Not all features are implemented by "cross-cuts." As the granularity of a feature scales, an individual "feature" might be encapsulated by a COM, .NET, or CORBA component [10]. How does AHEAD generalize to represent these features?
- The Principle of Uniformity imposes object model organizations on all artifacts. There are cases where *sequences* of artifacts, not *sets*, must be expressed and refinement operations on sequences (e.g., insert `x` before `y`, or insert `x` after `z`) must be considered. For example, XML documents are trees, but often the order matters in which children of a parent node are listed. More work is needed to define more precisely the concept of object model organizations with sequencing so that the generalization of the Principle of Uniformity covers ordered sequences as well.

10 Relationship to Other Work

AHEAD is a meld of ideas from several disciplines. To place our work in perspective, we first consider broad relationships to other paradigms in computer science and then relationships to specific prior work.

10.1 Perspective

10.1.1 Relational Query Optimization

We believe the future of software development lies in compositional programming and automated software engineering. The most successful example of both is *relational query optimization* (*RQO*). A relational query is specified in a declarative domain-specific language (SQL), a parser maps

it to an inefficient relational algebra expression, a query optimizer optimizes the expression automatically, and an efficient query evaluation program is generated from the optimized expression. RQO is a great example of *automatic programming*—transforming a declarative specification into an efficient program, and *compositional programming*—a program is synthesized from a composition of algebraic operators.

AHEAD is a generalization of the RQO paradigm so that both compositional programming and automated software development can be realized in many domains. AHEAD supports the paradigm of mapping declarative specifications (where users specify the features they want in their program) to an actual implementation. This is possible because programs are synthesized by composing modules that implement the required features and mapping feature specifications to equations has not been difficult [31], [7]. The novelty of AHEAD is that it models software domains as algebras, where features are operators. Particular programs are expressions—compositions of operators.

The power of the RQO paradigm is optimizing equational representations of programs using algebraic identities. The commutability of joins and the distributivity of projects over joins are examples in relational algebra. In 1998, we showed how equational representations of container data structures could be optimized using equational rewrites, thus demonstrating that programs in domains other than relational queries could benefit from equational optimizations. Our current research shows how tools can be automatically remodularized to minimize tool build times using equational manipulations. We believe that AHEAD lays the groundwork for equational manipulation and optimization of programs.

10.1.2 Algebraic Specifications

A formal approach to program development is *algebraic specifications* [18], [19], [53]. A specification is like a Java interface—it is a set of method signatures—but, more importantly, goes beyond Java by additional formal contractual obligations demanding that an implementation have certain properties. An implementation is said to *satisfy* a specification if it conforms to the specification's signatures and adheres to its contract. The semantics of a specification is then defined as all implementations that satisfy it. A specification leaves out implementation details and is thus not executable, but is intended to be more understandable and more easily written by humans. Step by step, it is made more concrete until there is only one possible implementation. At this point, there is a unique mapping from the specification to a program.

The two key points are: 1) Each *algebraic refinement step* preserves behavior without adding new functionality and 2) one can formally prove that this is so. Every implementation of a refined[5] specification is provably also an implementation of the original specification. The AHEAD concept of refinement is different, as AHEAD refinements usually add functionality to programs. Adding functionality to programs is called *extension* in algebraic specification parlance.

AHEAD and algebraic specification are complementary: AHEAD can profit from a rigorous formal foundation and from ideas on behavioral contracts (e.g., [38]). Whereas AHEAD encapsulates features, the unit of composition is more general in algebraic specification [53], [19], [14], [11]. On the other hand, algebraic specifications do not have AHEAD's tree-structured representation and support for multiple artifact types. Another key difference is that AHEAD units are *designed* to encapsulate implementations of reusable features; this is not required in work on algebraic specifications.

Because AHEAD does not deal with algebraic specifications in the traditional sense, the exact relationship is a subject of future work. A goal would be to extend AHEAD's composition operator to apply to algebraic specifications so that collectives, when composed, simultaneously construct a combined specification and its implementation. Further, a premise of AHEAD is that no single specification formalism can cover (or indeed should cover) all aspects of a program. AHEAD allows any number of related specifications to be composed. In this regard, AHEAD is less like algebraic specification and more like model-driven architectures [37], where a system is defined using multiple distinct and overlapping specifications.

10.1.3 Functional Programming

The idea that the architectural designs of programs can be given a functional form is implicit in many papers on software architecture. The core ideas of metaprogramming—programs are values and functions that transform these values—originated in functional languages. And the recursive application of apply operators to fold trees also has an elegant representation in functional languages.[6]

AHEAD shows how functional (or algebraic) descriptions of programs scale. Traditional metaprograms deal with the definition and composition of small code fragments (e.g., LISP quote and unquote); AHEAD, in contrast, deals with programs and program refinements on a *much* larger, hierarchical, multiclass scale. Although AHEAD is a "functional" model of programs, it is built around the object-oriented concepts of classes and inheritance, concepts that are not normally part of functional languages.

10.1.4 Feature Interactions

The selection of one feature may disable or enable the selection of other features. We have developed simple and efficient algorithms [3] to validate compositions of features automatically using Perry's "Light Semantics" [40]. This has been the predominant form of feature interactions that we have seen.

However, there are domains where generated software must have certified properties. It has been observed that properties of programs are often properties of individual features. When features are composed, we want assurances that these properties remain valid. Thus, an approach to the

5. There are other definitions for *refinement* in the literature [44], [12]. We use the most common one.

6. An equally strong claim for these points could be made for macro programming as well. We focus on functional programming in this section because the AHEAD model has many of the characteristics of functional languages.

synthesis of correct software is to certify properties of a feature and to certify that these properties hold after composition. Li et al. have developed special model checkers that show how the state space of programs can be reduced using a feature-oriented approach [34]. Their work may lead to composition operators for model-checking feature-oriented software.

The domain of telephony is one where feature interactions have been extensively studied. One approach, exemplified by Zave and Jackson [55], [29], is to independently define telephony features and their interface properties (e.g., activation conditions) in formal description languages such as Promela (used with the Spin model-checker [27]) and Alloy [28]. Using these descriptions, the goal of composition operators is to combine them into subsystem descriptions, taking into consideration precedences and priorities, to yield models that can be formally evaluated.

Prehofer [41], [42], also motivated by domains such as telephony where feature interactions are fundamental, developed a generalization of mixin inheritance [13], [21] that explicitly describes pairwise interactions as a "lifting" from one feature to another. A *lifting* is a set of code modifications that is applied when its associated interaction occurs in a mixin composition. For example, the resolution of mixin `A<B<C>>` would automatically incorporate the pairwise liftings from `C` to `B`, `B` to `A`, and `C` to `A`. Liftings appear to give AHEAD refinements more internal structure than what we have described and have exploited.

10.2 Other Work

Among the most advanced work on generators is that of *Model Integrated Computing* (*MIC*) at Vanderbilt [46]. 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 is 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 derived 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 provides a structure to modularize domains by features and a mechanism to synthesize applications by feature composition.

Domain-specific languages have long been recognized as a more efficient way in which to 1) specify applications and 2) integrate domain-specific analyses to validate DSL programs [51]. DSL usage is increasing and has been particularly successful in the specification of product lines [52]. AHEAD not only embraces the use of different general-purpose languages and DSLs as the primary means for specifying artifacts, but also advocates that programs written in these languages can be refined. Our refinement of state machines and equation files in AHEAD is an example. State machines are expressed in a DSL embedded in Java, while equation files are written in a standalone DSL. It is this scaling of refinements to code and noncode artifacts that is a distinguishing feature of AHEAD.

Aspect Oriented Programming (*AOP*) is a program refinement technology [33]. 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 function that is currently lacking in the AHEAD tool set. Gray et al have shown how aspects apply to noncode artifacts [22]. AHEAD shows how both code and noncode artifacts can be refined simultaneously in collection hierarchies.

Multidimensional Separation of Concerns (*MDSoC*) is another program refinement technology [39]. We have built GenVoca generators using `Hyper/J`. Features correspond to hyperslices, and GenVoca equations correspond to compositions of hyperslices. Further, MDSoC advocates that the techniques for assembling customized code from hyperslice compositions should also work for noncode artifacts as well. This conjecture inspired work on AHEAD. The contribution of AHEAD is a simple algebraic model that supports the MDSoC thesis.

11 Conclusions

We believe 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.

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 using 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 noncode—as a containment hierarchy and to treat containment hierarchies as values and 2) express feature refinements as functions that transform these values. 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 mathematical 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, most importantly, 3) keep generators based on stepwise refinement simple as the systems they synthesize scale in complexity.

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

ACKNOWLEDGMENTS

The authors thank Martin Wirsing, Alex Knapp, and Ira Baxter for discussions on algebraic specifications. They thank Melanie Kail and Mark Esslinger for their contributions to the design of AHEAD tools and Jim Browne, Stan Jarzabek, Dewayne Perry, and Roberto Lopez-Herrejon for their helpful comments on earlier drafts. They also appreciate the helpful comments of the referees. This work was supported in part by the US Army Simulation and Training Command (STRICOM) contract N61339-99-D-10 and Deutsche Forschungsgemeinschaft (DFG) project WI 841/6-1 "InOpSys."

REFERENCES

[1] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming," *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 Trans. Software Eng. Methodology,* Oct. 1992.
[3] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators," *IEEE Trans. Software Eng.,* vol. 23, no. 2, pp. 67-82, Feb. 1997.
[4] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages," *Proc. Fifth Int'l Conf. Software Reuse,* June 1998.
[5] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators," *IEEE Trans. Software Eng.,* vol. 26, no. 5, pp. 441-452, May 2000.
[6] D. Batory et al., "Achieving Extensibility through Product Lines and Domain-Specific Languages: A Case Study," *ACM Trans. Software Eng. and Methodology,* Apr. 2002.
[7] D. Batory, R.E. Lopez-Herrejon, and J.-P. Martin, "Generating Product lines of Product-Families," *Automated Software Eng.,* Sept. 2002.
[8] D. Batory, J. Liu, and J.N. Sarvela, "Refinements and Multi-Dimensional Separation of Concerns," *Proc. ACM SIGSOFT Conf. (ESEC/FSE2003),* 2003.
[9] I. Baxter, "Design Maintenance Systems," *Comm. ACM,* Apr. 1992.
[10] P.A. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, and D. Shutt, "Microsoft Repository Version 2 and the Open Information Model," *Information Systems,* vol. 24, no. 2, pp. 71-98, 1999.
[11] M. Bidoit and P.D. Mosses, "Common Algebraic Specification Language User Manual," *Lecture Notes in Computer Science,* IFIP Series, Springer-Verlag, 2003.
[12] "Logic Programming Synthesis and Transformation," *Proc. Ninth Int'l Workshop Logic-Based Program Synthesis and Transformation (LOPSTR '99),* A. Bossi, ed., Sept. 1999.
[13] G. Bracha and W. Cook, "Mixin-Based Inheritance," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 303-311, 1990.
[14] R. Burstall and J. Goguen, "Putting Theories Together to Make Specifications," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence,* pp. 1045-1058, 1977.
[15] R. Cardone and C. Lin, "Comparing Frameworks and Layered Refinement," *Proc. Int'l Conf. Software Eng.,* pp. 285-294, 2001.
[16] C. Clifton, G.T. Leavens, C. Chambers, and T. Millstein, "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 130-145, 2000.
[17] E.W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.
[18] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer, 1985.
[19] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer, 1990.
[20] I. Forman and S. Danforth, *Putting Metaclasses to Work.* Addison-Wesley, 1999.
[21] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and Mixins," *ACM Principles of Programming Languages,* pp. 171-183, 1998.
[22] J. Gray, T. Bapty, S. Neema, and J. Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Comm. ACM,* Oct. 2001.
[23] M. Griss, "Implementing Product line Features by Composing Component Aspects," *Proc. First Int'l Software Product Line Conf.,* Aug. 2000.
[24] W. Harrison and H. Ossher, "Subject-Oriented Programming (a Critique of Pure Objects)," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 411-427, 1993.
[25] W. Harrison, C. Barton, and M. Raghavachari, "Mapping UML Designs to Java," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* 2000.
[26] A. Hein, M. Schlick, and R. Vinga-Martins, "Applying Feature Models in Industrial Settings," *Proc. Software Product Line Conf. (SPLC1),* Aug. 2000.
[27] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual.* Pearson Educational, Sept. 2003.
[28] D. Jackson, I. Shlyakhter, and M. Sridharan, "A Micromodularity Mechanism," *ACM SIGSOFT Conf. (FSE/ESEC '01),* Sept. 2001.
[29] M. Jackson and P. Zave, "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services," *IEEE Trans. Software Eng.,* vol. 24, no. 10, pp. 831-847, Oct. 1998.
[30] J. Liu and D. Batory, "Automatic Remodularization and Optimized Synthesis of Product-Families," to appear.
[31] R.E. Lopez-Herrejon and D. Batory, "A Standard Problem for Evaluating Product line Methodologies," *Proc. 2001 Conf. Generative and Component-Based Software Eng.,* 2001.
[32] K.C. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis Feasibility Study," Technical Report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie-Mellon Univ., 1990.
[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. Ann. European Conf. Object-Oriented Programming,* pp. 220-242, 1997.
[34] H. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for Modular Feature Verification," *Proc. Conf. Automated Software Eng.,* 2002.
[35] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 97-116, 1998.
[36] S. McDirmid, M. Flatt, and W.C. Hsieh, "Jiazzi: New-Age Components for Old-Fashioned Java," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* 2001.
[37] S.J. Mellor, A.N. Clark, and T. Futagami, "Model-Driven Development," *IEEE Software,* no. 5, pp. 14-18, Sept./Oct. 2003.
[38] M. Nenninger and F. Nickl, "Implementing Data Structures by Composition of Reusable Components: A Formal Approach," *Proc. ICSE-17 Workshop Formal Methods Application in Software Eng. Practice,* M. Wirsing, ed., Apr. 1995.

[39] H. Ossher and P. Tarr, "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software," *Comm. ACM,* vol. 44, no. 10, pp. 43-50, Oct. 2001.
[40] D. Perry, "The Logic of Propagation in the INSCAPE Environment," *Proc. ACM SIGSOFT Conf.,* 1989.
[41] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," *Proc. Ann. European Conf. Object-Oriented Programming,* 1997.
[42] C. Prehofer, "Feature-Oriented Programming: A New Way of Object Composition," *Concurrency and Computation,* vol. 13, 2001.
[43] T. Reenskaug et al., "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems," *J. OO Programming,* vol. 5, no. 6, pp. 27-41, Oct. 1992.
[44] D. Sannella, "Algebraic Specification and Program Development by Stepwise Refinement," *Logic Program Synthesis and Transformation,* pp. 1-9, 1999.
[45] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Trans. Software Eng. Methodology,* Apr. 2002.
[46] J. Sztipanovits and G. Karsai, "Generative Programming for Embedded Systems," *Proc. Conf. Generative Programming and Component-Based Eng. (GPCE),* Oct. 2002.
[47] K.J. Sullivan and D. Notkin, "Reconciling Environment Integration and Software Evolution," *ACM Trans. Software Eng. Methodology,* vol. 1, no. 3, pp. 229-268, July 1992.
[48] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. Int'l Conf. Software Eng.,* 1999.
[49] Univ. of Texas Center for Agile Technology, "AHEAD Tool Documentation," 2002.
[50] M. VanHilst and D. Notkin, "Using Role Components to Implement CollaborationBased Designs," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 359-369, 1996.
[51] A. vanDeursen and P. Klint, "Little Languages: Little Maintenance?" *Proc. SIGPLAN Workshop Domain-Specification Language,* 1997.
[52] D. Weiss and C.T.R. Lai, *Software Product Line Engineering.* Addison-Wesley, 1999.
[53] M. Wirsing, "Algebraic Specification," *Handbook of Theoretical Computer Science,* pp. 676-788, 1990.
[54] W.A. Wolf, M. Shaw, P.N. Hilfinger, and L. Flon, *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.
[55] P. Zave, "An Experiment in Feature Engineering," *Programming Methodology,* pp. 353-377, 2003.

Don Batory holds the David Bruton Centennial Professorship at the University of Texas at Austin. He was an associate editor of the *IEEE Transactions on Software Engineering* (1999-2002), associate editor of the *ACM Transactions on Database Systems* (1986-1992), a member of the ACM Software Systems Award Committee (1989-1993; committee chairman in 1992), and program cochair for the 2002 Generative Programming and Component Engineering Conference. He has given numerous tutorials on product-line architectures, generators, and reuse and is an industry-consultant on product-line architectures. He is a member of the IEEE.

Jacob Neal Sarvela received the MA degree in mathematics from the University of California at Davis and the MS degree in computer sciences from the University of Texas at Austin, where he currently is a PhD student. His research interests are in generative programming, including its formalization and extension to build systems. He is a student member of the IEEE.

Axel Rauschmayer received the diploma in computer science from the University of Munich and wrote his diploma thesis in cooperation with the University of Texas at Austin. He was also one of the first three technical people behind Pangora, a shopping portal company that now powers the shopping pages of a wide range of European sites (among others, Yahoo, AOL, and Hotbot). Currently, he is a PhD student at the University of Munich. His main research interests are using graph-based knowledge representations for all aspects of software engineering, generative programming, and language design. He is a student member of the IEEE.